ELEVENTH ANNUAL

PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

October 18 - 20, 1993

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Preface . . . . . **vi**

Conference Officers/Committee Chairs . . . . . **vii**

Conference Planning Committee . . . . . **vii**

Presenters . . . . . **ix**

Exhibitors . . . . . **xii**

KEYNOTE
"The Acid Test for Quality" 1
Tim Lister, Atlantic Systems Guild, Inc.

SOFTWARE EXCELLENCE AWARD
"Core Teams at Teams Voice" 4
Laura Vernum, Active Voice

PLENARY SESSION

"High-Pressure Steam Engines & Computer Software" 10
Nancy Leveson, Computer Science & Engineering, University of Washington

PROCESS TRACK - October 19

"Motorola Drives Software Improvement" 13
Lynne Foster, Motorola

"Industry-wide Software Process Improvement" **39**
Wolfgang B. Strigel, Software Productivity Centre, Canada

"Brain Children Transform Chaos Manor into Productivity Palace" **48**
Sarah L. Sullivan, Ph.D., Indiana University–Purdue University

"Risk Management and Quality in Software Development" **58**
Ronald P. Higuera & David P. Gluch, Software Engineering Institute, CMU

"Implementing Software Process Improvement" **74**
Karen S. King, Credence Systems Corporation

"Wave Model for a Software Quality Process Program" **84**
Rebecca Joos & Ilona Rossman, Motorola

TESTING TRACK - October 19

"Mutation Analysis with the GRAIL System" **96**
Ishbel Duncan & Dave Robson, Durham University, England

"Selecting Functional Test Cases for a Class" **109**
John D. McGregor, Clemson University

"True Stories: A Year in the Trenches with MOTHER" **122**
Joe Maybee, Tektronix & Portland State University

"Test Control: An Alternative to SCCS/RCS Testing Suites" **135**
Owen (Rick) Fonorow, NCR Corporation

"Applying Statistical Testing to a Real-Time Embedded System" **154**
Dr. James A. Whittaker, Software Engineering Technology
Kaushal Agrawal, IBM Corporation

"A Mean Time to Failure Software Testing Methodology" **171**
Matt Siegel & Ian Ferrell, Microsoft Desktop Applications Division

ENGINEERING TRACK - October 19

"Implementing User Interface Consistency in a Distributed Organization" **185**
F. Beachley Main, Halliburton Energy Services Company

"A Case of Establishing a Coding Standard" **193**
Spass Stoiantschewsky & James Teisher, Credence Systems Corporation

"FOCS: A Classification System for Software Reuse" **201**
Jürgen E.W. Börstler, Aachen University of Technology, Germany

"A Return-of-Effort Model for Software Reuse" **212**
Thomas Grechenig & Stefan Biffl, Vienna University of Technology, Austria

"Test Suites for C++ Class Libraries" **223**
Dan Hoffman, University of Victoria, Canada

"Building Extensible Object-Oriented Systems" **234**
Jim Donahue, Documentum, Inc.

PROCESS TRACK - October 20

"Improving Software Quality Through Practical CMM Level Progression" **240**
Larry Cousin, 3M Health Information Systems

"A Software System Documentation Process Maturity Appproach to Software Quality **257**
Marcello Visconti & Curtis R. Cook, Oregon State University

"The FDA and Software for Medical Devices" **272**
Dennis Rubenacker, Noblitt & Rueland

"Assessment for ISO 9000-3 Compliance Using the SDI Method" **273**
Vladan Jovanovic & Dan Shoemaker, University of Detroit, Mercy

"Process vs. Product Assurance: Is There Really a Conflict?" **292**
John Joseph Chilenski, Boeing Commercial Airplane Group

"The Role of the Quality Group in Software Development" **293**
Douglas Hoffman, Software Quality Methods

MANAGEMENT TRACK - October 20

"Experiences with CSRS: An Instrumented Software Review Environment" **301**
Philip Johnson, University of Hawaii

"Quality Guided Programming: Integrating Code Reviews with Metrics Analysis of Code" **317**
Stefan Biffl & Thomas Grechenig, Vienna University of Technology, Austria

"Management Report: Metrics to Evaluate the Quality of Your Software" **328**
Michael Strange, KPMG Peat Marwick

"1992 Metrics Research Report" **343**
Nancy Spencer, Applied Business Technology

"Using the Poka Yoke Method for Improving Process" **344**
James Tierney, Microsoft Corporation

"Change Management in Small Companies" **354**
Don White, Cascade Solutions

ENGINEERING TRACK - October 20

"CITE: An Integrated Environment for Automated Testing" **361**
Peter Vogel, Convex Computer Corporation

"Improving Application Software Quality Using Motif-based Rapid GUI Prototyping" **374**
Roger Harrison & Larry Cousin, 3M Health Information Network

"Testing Graphical User Interfaces" **391**
Ellis Horowitz, Ph.D. & Zafar Singhera, University of Southern California

"Software Configuration Management Tools: The Next Generation" **411**
Steven King, Tektronix Laboratories

"Complexity in Interface Modeling: A Comparison of Two Structural Forms" **427**
Patrick Loy, Loy Consulting, Inc.

Index **438**

Proceedings Order Form . . . . . **Back Page**

PREFACE

Mark A. Johnson

Welcome to the Eleventh Annual Pacific Northwest Software Quality Conference. As we begin our second decade I am proud to see how our conference has blossomed from a small group of technical professionals who wanted to share ideas and learn new information into an internationally recognized gathering.

This year we have speakers from around the world to combine with the best of what is happening within industry across the United States. We have brought this all here to the Oregon Convention Center to share with the Pacific Northwest in a forum where software professionals can meet, exchange information, learn new ideas and acquire new skills.

Our keynote speaker is **Timothy Lister**, a leading international consultant on software quality and productivity. He is best known as the co-author of the book *Peopleware: Productive Projects and Teams*. His keynote address is entitled "The Acid Test for Quality," in which he discusses the organizational decisions that can either hinder or help software quality efforts.

We are pleased to publish these *Proceedings*, which contain the papers presented during the technical program. The programs represent current thinking and practice from the United States, Canada and Europe.

I would like to thank Sue Bartlett and Dick Hamlet, the Program Committee Co-Chairs, for putting in the hard work it takes to pull the program together. Their work is the basis for this Conference. Many thanks also go to the Abstract Selection Committee for their efforts in reviewing and selecting these papers from over 50 abstracts. Much thanks also to all members of the Program Committee for their time and energy in reviewing the draft papers from selected authors.

I would also like to thank the members of all the other committees, whose names are listed in the next section, for their contributions to the success of this Conference.

Finally, I want to express special thanks to Terri Moore of Pacific Agenda for being able to handle the organizational and administrative tasks while at the same time keeping all the committees on track and moving toward the successful presentation of the Conference.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Mark Johnson - President/Chair
Mentor Graphics

Steve Shellans - Vice President, Software Excellence Award
Moni Research

Hilly Alexander - Secretary
ADP

Ray Lischner - Treasurer
ParcPlace Systems

Sue Bartlett - Program Co-Chair
Tektronix, Inc.

Dick Hamlet - Program Co-Chair
Portland State University

Connie Ishida - Workshops Co-Chair
Mentor Graphics

Miguel Ulloa - Workshops Co-Chair
Mentor Graphics

James Mater - Exhibits
Technical Solutions, Inc.

G.W. Hicks - Publicity
TSSI, Inc.

Karen King - Birds of a Feather
Credence Systems Corporation

CONFERENCE PLANNING COMMITTEE

Chuck Adams
Tektronix, Inc.

John Burley
Flir Systems

Curtis Cook
Oregon State University

Margie Davis
ADP Dealer Services

Dave Dickmann
Hewlett-Packard

Michael Franks
IMS

Cynthia Gens
Solus Systems, Inc.

Michael Green

Bill Junk
University of Idaho

Ken Maddox
Software Association of Oregon

Peter Martin
Taligent, Inc.

Howard Mercier
Intersolv

Jerry Pitts
Credence Systems Corporation

Eric Schnellman
Boeing Commercial Aircraft Co.

George Tice

Jay Van Sant
Management Resources

Barbara Zanzig
NeoPath

PROFESSIONAL SUPPORT

John Bistolas
Bistolas and Associates

Betsy Cluff
Graphic Design

Pacific Agenda, Inc.
Conference Management

PRESENTERS

Kaushal K. Agrawal
IBM (AdStaR)
Dept: 79F/041-01
9000 S. Rita Road
Tucson, AZ 85744

Stefan Biffl
Dept. of Software Engineering
Technical University of Vienna
A-1040 Resselgasse 3/2/188
Vienna, Austria, Europe

Jürgen Börstler
Lehrstuhl fur Informatik III
Aachen University of Technology
Ahornstrabe 55, A-5100 Aachen
Germany

John Joseph Chilenski
Boeing Commercial Airplane Group
PO Box 3707 MS 6Y-07
Seattle, WA 98124-2207

Larry Dean Cousin
3M Health Information Systems
575 W. Murray Blvd.
PO Box 57900
Murray, UT 84157-0900

Jim Donahue
Documentum, Inc.
4683 Chabot
Pleasanton, CA 94588

I.M.M. Duncan
Centre for Software Maintenance
School of Engineering
and Computer Science
University of Durham
Science Laboratories, South Road
Durham, England DH1 3LE

Owen Richard Fonorow
NCR Corporation
1100 E. Warrenville Road
1W 2Z-464
Naperville, IL 60566

Lynne Foster
Motorola, Mobile Data Division
11411 Number 5 Road
Richmond, BC
Canada V7A 4Z3

Thomas Grechenig
Dept. of Software Engineering
Technical University of Vienna
A-1040 Resselgasse 3/2/188
Vienna, Austria, Europe

Roger Harrison
3M Health Information Systems
575 W. Murray Blvd.
PO Box 57900
Murray UT, 84157-0900

Ronald R. Higuera
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213

Dan Hoffman
University of Victoria
Computer Science Department
PO Box 3055
Victoria, BC
Canada V8W 3P6

Douglas Hoffman
Software Quality Methods
1449 San Tomas Aquino Road
San Jose, CA 95130

Ellis Horowitz
Computer Science Department
University of Southern California
Los Angeles, CA 90089

Philip Johnson
Dept. of Information
and Computer Sciences
University of Hawaii
Honolulu, HI 96822

Becky Joos
Motorola
6501 William Cannon Drive
Austin, TX 78735

Karen S. King
Credence Systems Corporation
9000 SW Nimbus Avenue
Beaverton, OR 97005

Steven King
Software Technology Research Lab
Tektronix Laboratories
PO Box 500, MS-50-662
Beaverton, OR 97077

Nancy Leveson
Dept. of Computer Science
and Engineering, FR-35
Sieg Hall 114
University of Washington
Seattle, WA 98195

Tim Lister
Atlantic Systems Guild
353 West 12th Street
New York, NY 10014

Patrick Loy
Loy Consulting, Inc.
PO Box 24648
Baltimore, MD 21214

F. Beachley Main
Sierra Geophysics, Inc.
A Halliburton Co.
11255 Kirkland Way
Kirkland, WA 98033

Joe Maybee
Tektronix, Inc.
PO Box 1000, MS-63-424
Wilsonville, OR 97070

John D. McGregor
Object Technology Group
Dept. of Computer Science
Clemson University
Box 341906
Clemson, SC 29634-1906

Dennis Rubenacker
Noblitt & Rueland
5405 Alton Parkway, #A530
Irvine, CA 92714

Dan Shoemaker, Ph.D.
College of Business Administration
University of Detroit, Mercy
4001 W. McNichols Road
Detroit, MI 48221-9987

Matt Siegel
Microsoft Corporation
Desktop Applications Division
One Microsoft Way
Redmond, WA 98052-6399

Nancy Spencer
Applied Business Technology
5600 Hillcrest, #1-L
Lisle, IL 60532

Michael Strange
KPMG Peat Marwick
345 Park Avenue
New York, NY 10154

Wolfgang B. Strigel
Software Productivity Centre
450-1122 Mainland Street
Vancouver, BC
Canada V6B 5L1

Sarah L. Sullivan, Ph.D.
Dept. of Computer Science
Indiana University
Fort Wayne, IN 46805

Jim Teisher
Credence Systems Corporation
9000 SW Nimbus Avenue
Beaverton, OR 97005

James Tierney
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Marcello Visconti
Computer Science Department
Oregon State University
Corvallis, OR 97331

Peter Vogel
Convex Computer Corporation
3000 Waterview Parkway
Richardson, TX 75083

Donald H. White, Jr.
Cascade Solutions
4650 NW Saint Helens Road
Portland, OR 97210

James A. Whittaker
Software Engineering Technology
601 S. Concord Street
Suite LLC
Knoxville, TX 37919

EXHIBITORS

Robert Philpott
Applied Business Technology
19508 SE 332nd Place
Auburn, WA 98002

Sabrina Rankin
Atria Software, Inc.
24 Prime Park Way
Natick, MA 01760

Blaine Bragg
Bender & Associates
PO Box 849
Larkspur, CA 94939

Cindy Mazzuchelli
Eastern Systems Inc.
PO Box 310
Hopkinton, MA 01748

Margaret Quigley
Interactive Development Environment
595 Market St., 10th Floor
San Francisco, CA 94105

Diana Jayes
KnowledgeWare, Inc.
3340 Peachtree Road NE, #1100
Atlanta, GA 30326-1050

Joan Brown
McCabe & Associates, Inc.
5501 Twin Knolls Road, #111
Columbia, MD 21045

Alex Ellis
Mercury Interactive Corporation
3333 Octavius Drive
Santa Clara, CA 95054

PNSQC
G.W. Hicks
PO Box 10142
Portland, OR 97210

Shawn Wall
Powell's Technical Books
33 NW Park Avenue
Portland, OR 97209

Lory Plumlee
PROCASE
2694 Orchard Parkway
San Jose, CA 95134

Teresa Harrison
SET Laboratories
PO Box 868
Mulino, CA 97042

Will Herman
Scopus Technology, Inc.
1900 Powell Street, #900
Emeryville, CA 94608

The Acid Test for Quality

Timothy Lister
The Atlantic Systems Guild, Inc.
353 West 12th Street
New York, New York 10014
(212) 620-4282

Introduction:

Six years ago, Tom DeMarco and I wrote a book titled, *Peopleware: Productive Projects and Teams*[1]. It still sells like hot-cakes, and is now available in six languages: English, Japanese, Dutch, Portuguese, German, and French. Unfortunately, one section of the book has received great response at the lip-flapping level, and little response at the action level. It's time for a short reminder. Since this talk is supposed to be the Keynote, let me humbly remind you that what I am talking about is not a side issue of the software business. If you are a manager, I am talking about your job — making it possible for your people to use their full intellectual horsepower on the complex problems that we ask them to solve.

Quality — If Time Permits, Encore.

The section of *Peopleware* that I am talking about was titled, "Quality — If Time Permits." Tom and I, in our travels to conferences around the globe, have heard loquacious speech after speech extolling the virtues of striving for ever-higher quality software. We then go to software development shop floors, and we hear a very different theme, "We have to get the software out yesterday." Somehow people don't want to discuss the inherent competition between the two strategies, if not downright incompatibility. I am concerned that for all our talk about the need for software of the highest quality, (here we all are at the Pacific Northwest Software Quality Conference), we still are not behaving as though we truly believe it.

Here it is, the Acid Test...

I have one question to ask you to determine if high quality is *really* the top priority at your organization, or just politically correct hot air?

> Does anybody do any serious Q.A. on the software project schedule?

Why your answer really matters...

Whether you are a large shop or a small group, most organizations practice some form of quality assurance on the products of their technicians. We now know that we need to worry to death the requirements, and build prototypes for confirmation of our understanding. We need to sweat the design. We need to have design and code inspections, if we are going to engineer a useful, acceptable product. But what about the sacred deadline? Is anybody even asking the question, "Given the deadline, can we expect to build a high quality product?"

Sad, but true, the deadline determines success...

Tom DeMarco and I give a seminar named Controlling Software Projects. In it we lead off with a statement we call The *Metric Premise*. It states:

> Rational competent men and women will seek to optimize any single, *observed* indication of success.

The key word here is *observed.* Nobody in our business will optimize unobserved indications of success, no matter how often they are told that they are important. If the organization does not measure, or in some other legitimate way determine, if a goal has been met, then the organization was never really serious about that goal. (This explains why we are always told that we need to build maintainable systems, and yet often we don't — maintainability is never measured.)

The simple, but brutal, truth is that for the vast majority of projects the only observed indication of success is the deadline. "It's the end of the year, did we deliver yet?"

- If the answer is yes, then the project is deemed a great success, and we'll have a big party with awards. (We'll just ship a few human debugger/fixers with the software to the client).
- If the answer is no, then the project is in deep trouble, and everybody better be working 12 hours a day, 7 days a week until we ship.

In either case, you don't even hear the word, quality. If quality is ever going to matter, then it must be measured. If quality is ever going to matter, we have to assure that projects are not so over-constrained by the deadline that high quality is impossible.

What gives???

I think of software projects as having three dimensions that are negotiable, project staff size, product functionality, and project duration. Staff size is simply how many competent folks do you get to help build a piece of software. Functionality is the exact definition of what functionality to what level of performance you have to supply in the product. Duration is the dreaded deadline again. How long do you get to build the system? If you are constrained on all three dimensions, like the majority of software projects, what gives? There can be only one thing that cracks — quality. "The three of us built all this software by the deadline," the team panted, holding their noses.

What should give?

It seems to me that software projects fall into two categories. Either they have *real* hard deadlines, in which case, functionality must be negotiable, or they must deliver a mandated set of features, in which cases deadline should be negotiable. Fixing functionality and duration is a very dangerous proposition. Given the sequential nature of software projects, and the high expense of communication overhead, adding staff to hold a deadline is dubious at best.

A Modest Proposal...

• Forget worrying about QA of the E-R diagram; QA the project plan and the proposed schedule. Answer the question, does this project plan and schedule place the quality of the product in jeopardy?

• Separate goals from estimates. State in writing what the goals of the project are, and how we will determine if we have attained these goals. As a completely separate exercise state the estimates; they are not the goals. (Think about it.)

• Prepare for problems, i.e. manage your risks. Before there is any sign of trouble, determine what you will do if you detect deadline slippage at various points in the project. (What you can do if you are 4 weeks behind schedule with 10 months to go is considerably different than what you can do if you are 4 weeks behind schedule with 10 weeks to go.)

A last word or two...

If you failed the acid test, don't impale yourself on your C compiler. Most folks fail too. On the other hand, there are enough organizations out there who do pass the test, that I know I'm right. Just be sure that if I come back next year, you'll pass.

---

[1] *Peopleware: Productive Projects and Teams*, Dorset House Publishing Co., NY, 1987,
1 - (800) DH - BOOKS. MC, Visa , AmEx; operators are standing by.

Core Teams at Active Voice

Abstract

Active Voice began exploring new methods of product development when projects were routinely running behind schedule, and the schedule slips were causing tension among staff. We have developed a team approach to product development, where members from the Development, Quality Assurance and Publications departments form self managed Core Teams. These Core Teams work on the project from its inception to its delivery to Production. This paper discusses in detail the Core Team approach used at Active Voice.

Biography

Laura Vernum has worked at Active Voice for seven years, and is currently Quality Assurance Manager. She is Co-Chair for the Washington Software Association QA SIG. Laura has a BA in Biology and English from Mount Holyoke College, and did Post Baccalaureate study in Computer Science at the University of Washington.

Introduction

Active Voice was confronted with problems two years ago when projects were running several months behind schedule. The company was growing rapidly, and in two years had more than doubled its staff from 40 employees to 90. During this time, we began having increasing difficulty meeting product schedules. As a very small company with a single product (less than twenty employees), the company functioned as a tightly knit project team. We had little difficulty meeting product schedules. But as we grew and "departmentalized," a lot of that team spirit was replaced with departmental affiliations. Projects that should have taken six months to complete were running months behind schedule. This resulted in tensions between departments.

When we studied the problem, we found that our method of product development was counter productive. First, Quality Assurance was getting involved too late in the development process, when the coding was complete and the requirements finalized. Design problems were being found in the Lab and Field Tests when they are expensive and time consuming to fix. It was often necessary to slip a schedule to fix a design problem. Second, each department was "bidding" the amount of time it would take them to complete their "part" of the project. No department, or individual, was responsible for the entire project. This made projects very difficult to manage. There always seemed to be pieces of the project that did not belong to any department and that would never get done. This resulted in a lot of finger pointing.

We needed more project ownership. Over the course of six months, we experimented with different methods of product development. We firmly believed that our staff wanted to work together and would do so if given the opportunity. Our approach was to give a small group of individuals a large but clearly defined set of objectives, and allow them to devote one hundred percent of their time to achieve these objectives. The team could divide up the work in what ever way was appropriate in order to achieve the objectives. We experimented with this approach over the course of six months, and the Core Team approach emerged.

Background

Founded in 1983, Active Voice makes PC based Voice Mail systems. There are three products in Active Voice's product line: Replay, Replay Plus, and Repartee. These are our low-end, mid-range and high-end products, respectively. Most of our product development centers around new features for these products, and new ways to integrate them with telephone systems, FAX machines and LAN's. We sell turn-key systems, where the software is bundled with a PC, or in a kit form, where the customer supplies the PC.

Among Active Voice's staff of 130, there are 21 Engineers in Development, 8.5 Engineers and 1 Field Test Manager in Quality Assurance, and 6 Technical Writers in Publications. There are 2 Product Managers and 9 Technical Support staff. The Development and

Publications departments report to the Vice President of Engineering, Quality Assurance reports to the President, and Technical Support and Product Management report to the Vice President of Marketing.

Core Teams

Core Teams are made up of at least one individual from each of the following departments: Quality Assurance, Development and Publications. The individuals are typically dedicated to the team full-time, meaning that they are not working on any other projects. One of the individuals on the Core Team serves as the Team Lead. Since any full-time Core Team member can be a Team Lead, the Lead can be a member of the QA and Publications departments, as well as Development. The Team Lead reports to the Engineering Manager for the duration of the project on project issues, and to their regular supervisor on any other issues. The Team Lead is a temporary role.

Team Lead Responsibilities

The Team Lead is responsible for working with the team to define a task list of all the activities required to complete the project, and for assigning these tasks. He or she is also responsible for creating the product schedule, and for notifying the Engineering Manager of any schedule slips. The Team Lead sets the pace for team communications, whether this be EMail, Voice Mail, or informal meetings, and maintains a history of the project.

Core Team Responsibilities

The responsibilities of the Core Team members are clearly defined. Members from Development are responsible for writing the Requirements Definitions and Design Documents, and for implementing the design. Quality Assurance is responsible for performing Usability Tests (typically performed on a prototype built from the Requirements Definition), and for conducting a Lab Test and a Field Test. Publications is responsible for writing the user documentation and on-line help. In the interest of meeting the schedule, team members are free to help each other and take on tasks not typically performed by a member of their department.

There are several responsibilities that are shared by all team members. These activities include training the departments that will support the product: Technical Support, Sales Support and Production. The team also writes the Product Release Plan, which is a detailed plan that coordinates all the release activities and ensures that the company is ready to start supporting the product. All team members are responsible for reviewing the Requirements Definitions. The team works with a Hardware Team to assess impacts the new product will have on the PC related hardware that we ship.

What gives???

I think of software projects as having three dimensions that are negotiable, project staff size, product functionality, and project duration. Staff size is simply how many competent folks do you get to help build a piece of software. Functionality is the exact definition of what functionality to what level of performance you have to supply in the product. Duration is the dreaded deadline again. How long do you get to build the system? If you are constrained on all three dimensions, like the majority of software projects, what gives? There can be only one thing that cracks — quality. "The three of us built all this software by the deadline," the team panted, holding their noses.

What should give?

It seems to me that software projects fall into two categories. Either they have *real* hard deadlines, in which case, functionality must be negotiable, or they must deliver a mandated set of features, in which cases deadline should be negotiable. Fixing functionality and duration is a very dangerous proposition. Given the sequential nature of software projects, and the high expense of communication overhead, adding staff to hold a deadline is dubious at best.

A Modest Proposal...

• Forget worrying about QA of the E-R diagram; QA the project plan and the proposed schedule. Answer the question, does this project plan and schedule place the quality of the product in jeopardy?

• Separate goals from estimates. State in writing what the goals of the project are, and how we will determine if we have attained these goals. As a completely separate exercise state the estimates; they are not the goals. (Think about it.)

• Prepare for problems, i.e. manage your risks. Before there is any sign of trouble, determine what you will do if you detect deadline slippage at various points in the project. (What you can do if you are 4 weeks behind schedule with 10 months to go is considerably different than what you can do if you are 4 weeks behind schedule with 10 weeks to go.)

A last word or two...

If you failed the acid test, don't impale yourself on your C compiler. Most folks fail too. On the other hand, there are enough organizations out there who do pass the test, that I know I'm right. Just be sure that if I come back next year, you'll pass.

---

[1] *Peopleware: Productive Projects and Teams*, Dorset House Publishing Co., NY, 1987,
1 - (800) DH - BOOKS. MC, Visa , AmEx; operators are standing by.

Core Teams at Active Voice

Abstract

Active Voice began exploring new methods of product development when projects were routinely running behind schedule, and the schedule slips were causing tension among staff. We have developed a team approach to product development, where members from the Development, Quality Assurance and Publications departments form self managed Core Teams. These Core Teams work on the project from its inception to its delivery to Production. This paper discusses in detail the Core Team approach used at Active Voice.

Biography

Laura Vernum has worked at Active Voice for seven years, and is currently Quality Assurance Manager. She is Co-Chair for the Washington Software Association QA SIG. Laura has a BA in Biology and English from Mount Holyoke College, and did Post Baccalaureate study in Computer Science at the University of Washington.

Introduction

Active Voice was confronted with problems two years ago when projects were running several months behind schedule. The company was growing rapidly, and in two years had more than doubled its staff from 40 employees to 90. During this time, we began having increasing difficulty meeting product schedules. As a very small company with a single product (less than twenty employees), the company functioned as a tightly knit project team. We had little difficulty meeting product schedules. But as we grew and "departmentalized," a lot of that team spirit was replaced with departmental affiliations. Projects that should have taken six months to complete were running months behind schedule. This resulted in tensions between departments.

When we studied the problem, we found that our method of product development was counter productive. First, Quality Assurance was getting involved too late in the development process, when the coding was complete and the requirements finalized. Design problems were being found in the Lab and Field Tests when they are expensive and time consuming to fix. It was often necessary to slip a schedule to fix a design problem. Second, each department was "bidding" the amount of time it would take them to complete their "part" of the project. No department, or individual, was responsible for the entire project. This made projects very difficult to manage. There always seemed to be pieces of the project that did not belong to any department and that would never get done. This resulted in a lot of finger pointing.

We needed more project ownership. Over the course of six months, we experimented with different methods of product development. We firmly believed that our staff wanted to work together and would do so if given the opportunity. Our approach was to give a small group of individuals a large but clearly defined set of objectives, and allow them to devote one hundred percent of their time to achieve these objectives. The team could divide up the work in what ever way was appropriate in order to achieve the objectives. We experimented with this approach over the course of six months, and the Core Team approach emerged.

Background

Founded in 1983, Active Voice makes PC based Voice Mail systems. There are three products in Active Voice's product line: Replay, Replay Plus, and Repartee. These are our low-end, mid-range and high-end products, respectively. Most of our product development centers around new features for these products, and new ways to integrate them with telephone systems, FAX machines and LAN's. We sell turn-key systems, where the software is bundled with a PC, or in a kit form, where the customer supplies the PC.

Among Active Voice's staff of 130, there are 21 Engineers in Development, 8.5 Engineers and 1 Field Test Manager in Quality Assurance, and 6 Technical Writers in Publications. There are 2 Product Managers and 9 Technical Support staff. The Development and

Publications departments report to the Vice President of Engineering, Quality Assurance reports to the President, and Technical Support and Product Management report to the Vice President of Marketing.

Core Teams

Core Teams are made up of at least one individual from each of the following departments: Quality Assurance, Development and Publications. The individuals are typically dedicated to the team full-time, meaning that they are not working on any other projects. One of the individuals on the Core Team serves as the Team Lead. Since any full-time Core Team member can be a Team Lead, the Lead can be a member of the QA and Publications departments, as well as Development. The Team Lead reports to the Engineering Manager for the duration of the project on project issues, and to their regular supervisor on any other issues. The Team Lead is a temporary role.

Team Lead Responsibilities

The Team Lead is responsible for working with the team to define a task list of all the activities required to complete the project, and for assigning these tasks. He or she is also responsible for creating the product schedule, and for notifying the Engineering Manager of any schedule slips. The Team Lead sets the pace for team communications, whether this be EMail, Voice Mail, or informal meetings, and maintains a history of the project.

Core Team Responsibilities

The responsibilities of the Core Team members are clearly defined. Members from Development are responsible for writing the Requirements Definitions and Design Documents, and for implementing the design. Quality Assurance is responsible for performing Usability Tests (typically performed on a prototype built from the Requirements Definition), and for conducting a Lab Test and a Field Test. Publications is responsible for writing the user documentation and on-line help. In the interest of meeting the schedule, team members are free to help each other and take on tasks not typically performed by a member of their department.

There are several responsibilities that are shared by all team members. These activities include training the departments that will support the product: Technical Support, Sales Support and Production. The team also writes the Product Release Plan, which is a detailed plan that coordinates all the release activities and ensures that the company is ready to start supporting the product. All team members are responsible for reviewing the Requirements Definitions. The team works with a Hardware Team to assess impacts the new product will have on the PC related hardware that we ship.

Project Team

The Core Team is a subset of a larger Project Team. The other members of the Project Team are the Product Manager (a member of the Marketing department), the Field Test Manager, the Production Engineer and a representative from the Technical Support department. Each of these individuals participates on the team on a part-time basis. They are responsible for reviewing the Requirements Definitions and any prototypes that are developed. The Product Manager and the Field Test Manager may belong to several teams at one time.

The Product Manager is responsible for doing the initial product research prior to the project's inception and for coordinating the activities of departments that are not on the project team. This includes working with Corporate Communications on press releases, brochures, dealer announcements, and coordinating specialized training for Technical Support, Sales Support and Production.

The Field Test Manager is responsible for finding Field Test sites, for contacting those sites weekly throughout the Field Test, and for bringing problems to the attention of the Core Team. Core Team members may assist the Field Test Manager in contacting sites and resolving problems during the busy parts of the Field Test.

The Technical Support representative is responsible for gaining expert knowledge on the product prior to its release, and for providing the Technical Support department with the documentation needed to support the product.

The Production Engineer is responsible for gaining in depth knowledge on the product prior to its release, and for preparing the Production Department for the release of the product. This is done by creating all the necessary procedures and checklists to build, test and ship the product.

Team Identity

Team identity bonds the team, and helps give the members a sense of ownership. Every project is officially "kicked off," with all the Core Team members, their supervisors and senior management present. The project objectives and roles of each of the team members are discussed. A project document clearly defines these objectives and roles, and is distributed to each team member.

Many of the teams adopt a team name that is loosely connected to some aspect of the project. These names are used in-house only, and some names that have been used are Passport, Magellan, Gemini, Phoenix to TeLANosaurus REX. There is a friendly competition among teams for stylish logos, which are often displayed on the office doors of team members as well as on cover sheets of intra-team communications.

Teams keep in close communication with each other through Voice Mail, EMail, regularly scheduled meetings, or most commonly, informal visits to each other's offices. In Development, team members share offices with each other, moving offices when the team's project is completed.

The Role of Management

The supervisors of the individuals on Core Teams have formed a Product Steering Committee that meets weekly to discuss the progress of the teams. This group is comprised of six managers from the Quality Assurance, Development, Product Management and Publications Departments. The group designs the teams together, attempting to achieve a balance of personalities and experience, and selects a project lead. When projects are completed, the Steering Committee reviews the project, discussing which aspects went particularly well, and aspects that need to be improved.

Current challenges

One of our biggest challenges is bringing the Core Team approach to the smallest projects. The Core Team approach has worked on projects that range from one month to one year, but has been more challenging to implement on very small projects that range from a couple of days to a week. We have had success in grouping together into a single team all projects related to qualifying third party products, such as PC's, hard drives, and new versions of MS-DOS. This team is called the Platform Team and has a full-time Hardware Engineer from Development, a half-time QA Engineer and a half-time Technical Writer from Publications. The team works together to coordinate all the projects, many of which can come up unexpectedly (such as when a vendor discontinues a product, or releases a new revision). This team has worked extremely well.

In the software area, it has been a little more difficult. Spur of the moment software projects, such as a special enhancement for a customer, or a defect fix, often require specialized knowledge of some aspects of our products. Three different projects over the course of a month may require different staff from each of the Quality Assurance and Engineering departments, because they require specific knowledge about an aspect of one of our products. For such short term projects, it often most efficient to use the more expert staff person, rather than take the time to train a new person. To achieve this, we often need to "borrow" an individual who is already assigned to a Core Team for short periods of time, ranging from a few hours to a couple of days. We hope to find a better way to handle very small software projects in the coming year.

Impact of Core Teams at Active Voice

Since we have been using Core Teams, we have seen a great increase in projects being completed on-time. Prior to the Core Team approach, most projects ran more than 2 months behind schedule. Now, most projects are completed on time, with occasional 1-2 week slips, and in one case a month slip.

The Core Team approach has also resulted in a great increase in camaraderie between staff in the Quality Assurance, Development and Publications departments. Tension between these departments is virtually non-existent. There has been a large increase in ownership and leadership, as more and more responsibility is put into the hands of the team members. Quality Assurance staff are much happier being members of a Core Team and participating actively in the product development process.

HIGH-PRESSURE STEAM ENGINES

AND COMPUTER SOFTWARE

Nancy G. Leveson
Computer Science & Engineering
University of Washington

leveson@cs.washington.edu

Even though a scientific explanation may appear to be a model of rational order, we should not infer from that order that the genesis of the explanation was itself orderly. Science is only orderly after the fact; in process, and especially at the advancing edge of some field, it is chaotic and fiercely controversial.

-- William Ruckelshaus

Great inventions are never, and great discoveries are seldom, the work of any one mind. Every great invention is really either an aggregation of minor inventions, or the final step of a progression.

-- Robert H. Thurston

A History of the Growth of the Steam Engine (1883)

Boiler technology lagged behind improvement in steam engines themselves.

- Avoid equating humans with machines and ignoring the cognitive aspects of our field.
- Proceed with caution, e.g., maintain hardware backups
- Keep things simple

There was little scientific understanding of the causes of boiler explosions.

- Changing from an art to a science requires accumulating and classifying knowledge.
- Need to validate and assess our hypotheses using scientific principles.
- Theoretical foundations can provide:

  Criteria for evaluation
  Means of comparison
  Theoretical limits and capabilities
  Means of prediction
  Underlying rules, principles, and structure

The safety features designed for the boilers did not work as well as predicted because they were not based on scientific understanding of the causes of accidents.

- Avoid proof by labelling, proof by definition, and other unscientific practices.
- Need to develop a foundation of knowledge about human error in software development.

The introduction of safety devices for steam engines was inhibited not only by the lack of underlying scientific knowledge about boilers, but also by a narrow view of attempting to design a technological solution without looking at the social and organizational factors involved and the environment in which the device is used.

- Need to consider the environment in which the software will be used including human factors.

- Technical solutions for safety can be cancelled out by organizational and management factors.

The operators of steam engines received most of the blame for accidents, not the designers or the technology.

- Software engineers need to take human factors more seriously.

- Human engineering experts should be involved in the design of safety-critical software interfaces.

The early steam engines had low standards of workmanship, and engineers lacked proper training and skills.

- Need to sort out training, licensing, and standards issues

- Increased government regulation is inevitable unless we take steps first to ensure safety in critical software and technical competence in those who build it.

MOTOROLA

Using the Motorola Quality System Review (QSR) to Drive Software Improvement

Lynne Foster
Motorola Wireless Data Group
(formally Mobile Data Division)
11411 Number 5 Road
Richmond B.C.
V7A 2Z3

604-241-6427

Abstract

For over 10 years, Motorola has used a 5-day audit called the Quality System Review (QSR) to drive constant improvement in its processes and products. A key area of the audit focuses on software development and quality.

This paper describes the QSR format, the evaluation criteria, how MDD prepared for its first QSR, and the benefits and problems MDD experienced as a result of the audit. It also summarizes the similarities and differences between the QSR and the SEI Assessment.

Biographical Sketch

Lynne Foster has worked at Motorola for 5 years. She has a B.Sc. in Computing Science and Mathematics from Simon Fraser University, Burnaby B.C.. She has worked in all aspects of software engineering including designing, programming, customer support, project management and department management. She has also taught Computer Science courses at several post-secondary institutions around British Columbia.

As the MDD Software Quality Assurance person, Lynne was responsible for preparing MDD's software engineering community for the Quality System Review (QSR). She has also conducted two QSR audits and one SEI assessment on other Motorola divisions and is currently preparing MDD (now the Wireless Data Group-WDG) for an SEI assessment in early 1994.

Using the Motorola Quality System Review (QSR) to Drive Software Improvement

A Quality System is defined as: "The collective plans, activities and events that are provided to ensure that products, processes and services will satisfy given customer needs."

1. QSR Overview

1.1 Introduction

Since 1981, Motorola has focused on quality and total customer satisfaction. Aggressive goals set by Motorola Corporate in Schaumburg have demanded a more sophisticated Quality System from all Motorola divisions, including the Mobile Data Division located in Richmond, B.C. Canada, and Bothell, Washington.

The key method used by Motorola to assess the Quality System of each division is the **Quality System Review (QSR).**

The QSR is a mandatory 5-day audit conducted biennially on all divisions of Motorola. The audit team is made up of 5-7 Motorolans from divisions other than the one being audited. The audit team travels to the audited division's site, receives background information on the division, interviews close to 100 employees from all levels of the organization, and then prepares a detailed report of the findings. Not only is the QSR a formal assessment of a division's Quality System, but it is a terrific opportunity for Motorolans to learn from each other. The QSR was a big contributor to Motorola winning the Malcolm Baldrige National Quality award in 1988.

1.2 The QSR Guidelines

A half-inch thick Guidelines document details how the audit is conducted, and what the auditors look for. This QSR Guidelines document has become the bible for how Motorola runs its businesses.

The QSR Guidelines contain 10 subsystems:

- Management
- Supplier Control
- Quality Data Programs
- Equipment Calibration
- Human Resources
- New Product Development
- Process Operations
- Problem Solving Techniques
- Customer Satisfaction
- Software Quality Assurance

Early versions of the QSR Guidelines did not have a software subsystem (indicating software's perceived importance to Motorola 10 years ago). Now that software is critical to Motorola's survival, the software subsystem is actually the largest and most detailed subsystem. Subsystem 10, Software Quality Assurance, is the main focus of this paper.

The QSR is not confidential to Motorola; Copies of the Guidelines are available to anyone: Motorola's suppliers, customers and competitors. Motorola believes that the high-tech industry can only benefit by all companies improving their processes, products and customer focus.

1.3 Evaluation Criteria

The QSR is extensive. Each subsystem has about 10 questions, giving over 100 questions in total to be scored as **Poor, Weak, Fair, Marginally Qualified, Qualified** and **Outstanding.** Motorola requires a score of Qualified for each question.

Appendix A shows the scoring sheet with the 12 questions for the Software Quality Assurance subsystem. A score is determined for each question, and an **X** is put in the appropriate column. Each question's score is multiplied by a weighting that is assigned by the audited division's Quality department. The weighting indicates how applicable the question is to the audited division's business. All subsystem scores are rolled up into an overall QSR score.

The twelve Software Quality Assurance subsystem questions are further broken down into sub-questions giving 60 individual software questions to be evaluated from Poor to Outstanding. Appendix B shows all 60 sub-questions.

There are three important dimensions used when evaluating each question: **Approach, Deployment** and **Results.** For a division to be

rated as Qualified or Outstanding, it must show it has all three components for each question. Having results without a documented and repeatable approach rates low, as does having an approach without measurable results.

The following chart shows the relationships between the ratings (**Poor, Weak, Fair, Marginally Qualified, Qualified and Outstanding**) and the evaluation dimensions (**Approach, Deployment and Results.**)

QSR General Scoring Matrix

	Evaluation Dimensions		
	Approach	**Deployment**	**Results**
Poor	• No System / Process Evident • No Management Recognition of Need	• None	• Ineffective
Weak	• Beginnings of a System/Process • A few factors in place • Management has begun to recognize the need	• Fragmented • Deployed in some areas of the business	• Spotty Results • Some Evidence of output
Fair	• Direction for system/process defined • Wide but not complete support by management	• Less fragmented • Deployed in some major areas of the business	• Inconsistent but positive results in areas deployed
Marginally Qualified	• A sound system/process in place with evidence of prevention activities	• Most major areas of the Business • Mostly consistent	• Positive measurable results in most major areas • Some evidence that results are caused by approach
Qualified	• Well designed/proven system/process which is prevention based with evidence of refinement and improvement and renewal • Majority of management is proactive • Total management support	• Pervasive and consistent across all major areas of the business	• Evidence that efforts are successful • All requirements fulfilled • Demonstrated positive and sustained results
Outstanding	• Exceptional well defined, innovative system/process that anticipates customer needs • Management provides zealous leadership • Recognized even outside the company	• Pervasive and consistent across all major areas of the business; both internal & external	• Requirements exceeded • World class results • Counsel sought by others

Appendix C shows a sample scoring matrix for one of the software questions. In the QSR Guidelines, there is one scoring matrix for each software sub-question.

The auditors' challenge is to interview the audited division's employees and use these matrices to fairly and accurately determine the division's Quality System. The auditors prepare a detailed report which gives the score for each question within each subsystem. The report also describes areas in which the division shows strengths, and areas the division has opportunities for improvement. The auditors present the report to the audited company's Senior Management and employees.

1.4 After the Audit

After the QSR audit is complete, the audited division has 3 months to prepare action plans for each question receiving a rating less than Qualified. These action plans detail how the division is going to use the next 2 years to drive the deficient areas of its Quality System to be at least Qualified.

The report prepared by the auditors and the action plans prepared by the audited company are presented to the Motorola Corporate Quality Council in Schaumburg.

The Council reviews the plans ensuring each question rated less than Qualified has an owner and a detailed plan with key deliverables and completion dates. If the division did not meet the Marginally Qualified rating, another QSR audit is scheduled for the next year; otherwise the next QSR audit is scheduled in 2 years. The follow-up QSR audits ensure a continuous commitment to improvement.

2. MDD's QSR

2.1 Preparing for MDD's QSR

MDD spent a year preparing for its first QSR audit, which was held February 8-12, 1993.

In February 1992, a Team Leader for each of the ten subsystems was selected. These Team Leaders were the people who lived and breathed the QSR for a year. A Sponsor from the Senior Management level was also selected for each subsystem.

Since the QSR Guidelines require a Qualified rating for each subsystem, the Team Leaders decided to find out where MDD rated. We conducted a self-assessment of MDD's processes using the QSR Guidelines. The results were cause for concern! Many of the activities in the Guidelines were being done, but were not formalized or documented. Other activities weren't being done as well as they should have been. Some were just not being done.

From the self-assessment, the Team Leaders identified key improvement initiatives and Champions to be responsible for them. Some initiatives could be accomplished in time for the actual QSR audit; others were long-term initiatives that would take years of effort to implement. Since we couldn't do everything we needed to, initiatives were prioritized considering criteria like: Biggest bang for the buck; biggest risk if it wasn't done; and impact on customer satisfaction.

The Team Leaders kept MDD focused on the initiatives, and in many cases were the Champions. Sponsors provided resources and visibility to the QSR issues. QSR projects were reviewed at monthly meetings with the same visibility as revenue-generating projects.

Some of the software improvement initiatives MDD focused on included:

- Documenting and institutionalizing a Product Life Cycle into which the Software Life Cycle fits. This ensures software is developed as part of a system with marketing, hardware, manufacturing and service.

- Documenting existing processes and results.

- Preparing guidelines and templates for Software Quality Assurance Plans, Software Project Management Plans and Software Requirements Specifications using the ANSI/IEEE standards.

- Collecting and reporting metrics based on when defects were found, how long they were taking to be closed, etc. Metrics were reported each month at Senior Management reviews. An Excel Macro program was written to automatically generate the charts.

- Implementing a documented, formal process that controlled software from the time Software Engineering finished unit testing, until the software was available from the off-site manufacturing facility.

- Implementing a Technical Ladder to allow Software Engineers career growth without having to enter a management role.

- Preparing 5 year plans that included quality goals, SEI Maturity goals, and technology strategies for our development processes and products.

- Documenting how software subcontractors are qualified and managed.

- Implementing a program to patent innovative software ideas.

- Forming a team to bring Configuration Management under control.

- Forming a tools and technology group.

A month before the audit, we collected all the information we had that supported what we were doing, and prepared Evidence Books for each subsystem. We then selected people from all parts of the company to be interviewed. We had 18 Software Development Engineers and Managers scheduled for interviews; over 100 interviews in total were scheduled.

We then waited for the QSR Audit team to arrive.....

2.2 The Week of the QSR

The actual week of the QSR was extremely busy. Interviews generally lasted 50 minutes, so auditors could interview up to eight people a day.

The following is a typical schedule auditors use:

	Morning	Afternoon	Evening
Sunday			Team arrives.
Monday	Introductions, company overview, tour.	Interviews.	Review evidence and notes.
Tuesday	Interviews.	Interviews.	Review more evidence and more notes.
Wednesday	Interviews	Interviews	Review all evidence and all notes. Draft results. Drink lots of coffee.
Thursday	Draft report. Schedule extra interviews if required.	Finalize report. Prepare presentation.	Verify results with the audited division's Quality Department. Finalize edits. Sleeeeep.
Friday	Present report to the audited division's Senior Management and employees.	Relax. Fly home. Sleeeeep.	

2.3 Benefits

Although MDD's results are confidential, the many tangible benefits of the QSR audit aren't. These tangible benefits include:

- MDD got the kick it needed to get going on an improvement program. Because the QSR is mandatory for all of Motorola and it is tied into Senior Manager's (and many other people's) performance reviews, it was easy to get buy-in.

- Morale increased as employees saw an investment in improving the processes used to develop products. Knowing the QSR and the identified action plans are reviewed by Motorola Corporate ensures they will continue and not be just short-lived initiatives. People became more enthusiastic about change, and more pro-active because the QSR created an environment focused on improvement.

- Improvement projects that started before the QSR audit are continuing with good momentum. New projects have been started up since the QSR. Large projects, like having a documented Configuration Management system with supported tools, have Senior Management support and visibility. Configuration Management had many false starts over the last few years, but it is now coming together because of the visibility it received in the QSR.

- 450 people came together for a focused cause. Almost everyone in the division was involved in some way.

- Many parts of the company were effectively re-organized around the principles of the QSR. For example, MDD's first SQA position was created in preparation for the QSR, a Configuration Management department was formed, and a Tools and Technology group is being formed.

- Communication improved significantly throughout the organization because standards were documented and the appropriate people were brought together to create, mistake proof and document their processes. People began working together with a proactive and defect-prevention attitude.

- Knowledge was shared between divisions. The SQA auditors took home a "care package" of the best things MDD was doing. I have participated in other division's QSR audits and brought back many documents that can be implemented at MDD much quicker than if we created them from scratch.

- MDD learned more about Motorola's culture (MDD was originally a small company called MDI that became part of Motorola in 1988.)

2.4 Pitfalls

Of course, not everything was rosy. MDD had problems which included:

- Some improvement projects that were initiated eventually floundered and failed with lack of resources and commitment. The successful projects were those that were someone's job, rather than an additional project to be done in someone's "spare time."

- Although Senior Management sponsors were assigned, it was sometimes difficult to get their attention and constant commitment. Again, successes were achieved when the improvement initiatives were part of the Senior Manager's day-to-day job.

- Some improvement projects suffered the usual project problems: poorly defined requirements, poor planning and tracking, lack of resources, etc. (i.e. they were run like SEI Level 1 projects.)

- There were -- and always will be -- cynics and critics of any improvement program. The challenge was to ensure people understood and obtained the benefits for their efforts.

3.Summary

The QSR is Motorola's method for ensuring all its divisions constantly improve their processes and products, and strive to be a world-class software company. Although it takes much effort to prepare for a QSR

audit, the benefits are well worth it. The QSR would not have been successful without the commitment of Motorola, the MDD Senior Managers and every MDD employee.

Without a QSR to constantly push us, MDD would not have accomplished the improvements it did during 1992 and 1993, and would not have the detailed plans it has for the next 5 years. Knowing that there will be a QSR audit scheduled every two years ensures MDD will not lose its momentum for improving its processes and products.

Because the QSR is not proprietary, other companies are welcome to use the QSR Guidelines as the basis for their own Quality System. Words of Wisdom: You will not be remotely successful unless the QSR is driven constantly from the top of the organization, like it is at Motorola.

Appendix - A Scoring Worksheets

Evaluation Work Sheet: Software Quality Assurance

ORGANIZATION: DATE: SUBSYSTEM: 10 - Software Quality Assurance		FACTOR RATING (R)						APPLICABILITY	SCORE
		POOR	WEAK	FAIR	MARG. QUAL.	QUALIFIED	OUTSTANDING		
NO.	DESCRIPTION	0	2	4	6	8	10	(A)	(RxA)
10.1	Is an approved, documented process used to guide the development and maintenance of all software that impacts Total Customer Satisfaction?								
10.2	Are software project planning and control mechanisms in place and followed?								
10.3	Is software developed as part of a total system using a phased development approach, intermediate deliverables, and review and approval based on entry and exit criteria?								
10.4	Is software developed in support of documented (formal, written, approved, updated, and available) requirements with conformance to these requirements verified?								
10.5	Is software developed and maintained under documented plans for configuration management and change control, including installation and customer configuration?								
10.6	Is software developed using proper tools and documented, approved procedures for security and information recovery, including disaster protection?								
10.7	Does software undergo system/acceptance testing by individuals or organizations not directly involved in the design or implementation of the product being tested? Does testing reflect customer usage?								

November, 1992

Evaluation Work Sheet: Software Quality Assurance

ORGANIZATION: DATE: SUBSYSTEM: 10 - Software Quality Assurance		FACTOR RATING (R)						APPLICABILITY	SCORE
		POOR	WEAK	FAIR	MARG. QUAL.	QUALIFIED	OUTSTANDING		
NO.	DESCRIPTION	0	2	4	6	8	10	(A)	(RxA)
10.8	Are there established goals for software quality including Six Sigma performance as the overall goal? Do the measurement systems provide tracking of progress towards these goals as well as highlight quality issues from the customer perspective?								
10.9	Does the quality assurance organization act as a customer advocate in software matters by assuring conformance to customer requirements and specifications and proper execution of the approved development process?								
10.10	Is there a mechanism used to ensure continuous software development process improvement?								
10.11	Is there a capability improvement program in place for all software organizations, including deployment and assessment of training?								
10.12	Is the process used by software subcontractors under control, and is conformance to requirements of subcontracted software verified?								
								SCORE	
November, 1992							Subsystem Rating (Score/10)	100	

Appendix - B Scoring Reference Book Questions and Themes

SOFTWARE SCORING REFERENCE BOOK QUESTIONS AND THEMES

1. ***Is an approved, documented process used to guide the development and maintenance of all software that impacts Total Customer Satisfaction?***

   A. Existence of a documented process approved by SQA with designated owner — Score:___

   B. The approved process covers all types of software and is followed — Score:___

   C. Conformance of existing process to the Motorola Quality Policy for Software Development — Score:___

   QUESTION FACTOR RATING (R) (average of theme scores): ____

2. ***Are software project planning and control mechanisms in place and followed?***

   A. Use of planning procedures and tools — Score:___

   B. Software Development Plans (including Software Quality Assurance Plans) for projects — Score:___

   C. Selection of appropriate methods based on experience — Score:___

   D. Execution and tracking of the software development projects against plans — Score:___

   E. Control mechanisms (milestones, status reviews, etc.) and tools are utilized — Score:___

   F. Risk management and contingency planning are performed — Score:___

   QUESTION FACTOR RATING (R) (average of theme scores): ____

3. ***Is software developed as part of a total system using a phased development approach, intermediate deliverables and review and approval based on entry and exit criteria?***

   A. Software issues identified and quickly resolved — Score:___

   B. All necessary software project phases are included — Score:___

   C. Defined process deliverables, including documentation — Score:___

   D. Existence of standards for developing quality software with SQA involvement & approval — Score:___

   E. Use of formal reviews for all phases — Score:___

   F. Approval procedure based on entry and exit criteria — Score:___

   QUESTION FACTOR RATING (R) (average of theme scores): ____

4. ***Is software developed in support of documented (formal, written, approved, updated and available) requirements, with conformance to these requirements verified?***

   A. Documented requirements with updates — Score:___

   B. Software verification activity (conformance of output to input) — Score:___

   C. Validation and traceability of requirements to design and code — Score:___

   D. Existence of a mechanism for ensuring requirement coverage during testing — Score:___

   QUESTION FACTOR RATING (R) (average of theme scores): ____

5. ***Is software developed and maintained under documented plans for configuration management and change control, including installation and customer configuration?***

A. Software Configuration Management process is followed Score:___

B. Software configuration management system is effective Score:___

C. Releases made in a controlled way, over time (including documentation) Score:___

D. Mechanism for change control with effective feedback to the change requestor Score:___

E. Change notification for project participants and customers Score:___

F. Customer notification for software installation and customer configuration prior to shipment Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

6. ***Is software developed using proper tools and documented, approved procedures for security and information recovery, including disaster protection?***

A. Availability and use of proper security procedures for protection of the software Score:___

B. Mechanism for ensuring that the security procedures are used and updated appropriately Score:___

C. Archival and recovery process is documented, effective, and followed Score:___

D. Enforcement ensures compliance Score:___

E. Validation of security procedures and back-up mechanisms Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

7. ***Does software undergo system/acceptance testing by individuals or organizations not directly involved in the design or implementation of the product being tested? Does testing reflect customer usage?***

A. Independent software test activity Score:___

B. Tester participation in early phase reviews Score:___

C. Early test plan creation Score:___

D. Testing similar to actual customer operating conditions is conducted, e.g., customer test lab Score:___

E. Mechanism for ensuring testing is representative of customer operational profiles Score:___

F. Effectiveness of software testing evaluated and tracked Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

8. ***Are there established goals for software quality including six sigma performance as the overall goal? Do the measurement systems provide tracking of progress towards these goals as well as highlight quality issues from the customer perspective?***

A. Quantitative quality goals set and tracked Score:___

B. Quantitative productivity goals set and tracked Score:___

C. Quality issues from the customer perspective and process improvement actions Score:___

D. Customer satisfaction analysis and actions Score:___

E. Use of software metrics for tracking progress towards the goals Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

9. *Does the quality assurance organization act as a customer advocate in software matters by assuring conformance to customer requirements and specifications and proper execution of the approved development process?*

A. Independent Software Quality Assurance function Score:___

B. Software Quality Assurance role defined and followed Score:___

C. SQA acting as a customer advocate when quality issues arise Score:___

D. Software delivery criteria defined and met before delivery Score:___

E. SQA participation in metric tracking of deliverables, software quality & customer satisfaction Score:___

F. Visibility to management in situations of non-conformance Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

10. ***Is there a mechanism used to ensure continuous software development process improvement?***

A. Software process function has owner, is followed, and is effective Score:___

B. Use of SEI assessments for process improvement Score:___

C. Analysis and feedback of information about the software development process Score:___

D. Existence of standards for developing quality software with SQA involvement and approval Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

11. ***Is there a capability improvement program in place for all software organizations, including deployment and assessment of training?***

A. Capability and skills growth for all software positions Score:___

B. Effective software training Score:___

C. Suitable work environment for software professionals Score:___

D. Development environment contains effective tools which are utilized Score:___

E. Longevity, satisfaction and turnover are tracked and are satisfactory Score:___

F. Software re-use demonstrated Score:___

G. Acquisition of new software technology Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

12. ***Is the process used by subcontractors under control and is conformance to requirements of subcontracted software verified?***

A. Software subcontractor process control contains same rigor as internal software process control Score:___

B. Conformance of subcontracted software to requirements Score:___

QUESTION FACTOR RATING (R) (average of theme scores): ____

Appendix - C Sample Software Scoring Matrix

QUESTION 10.2: Are software project planning and control mechanisms in place and followed?

THEME	POOR (0)	WEAK (2)	FAIR (4)	MARGINALLY QUALIFIED (6)	QUALIFIED (8)	OUTSTANDING (10)
Use of Planning Procedures and Tools	No planning procedures or tools are used on software projects. Their potential is not recognized.	Some software-related planning procedures and tools are under evaluation. Isolated projects have started to use them. Estimation, work flow analysis and adequate support resources are unlikely to be in place.	Planning procedures and tools are used partially on a few software projects. These may include project parameterestimation, work breakdown structures, dependency analysis, and provision for support resources. Some projects are collecting historical data that will be used for planning purposes in future projects.	Common planning procedures and tools are consistently used on most software projects, including estimation models, work breakdown structures, dependency analysis and adequate software support resources. Local data bases with historical data from projects have been established, and are used in planning future projects.	Common planning procedures and tools are consistently used for all software projects. Estimation, work breakdown structures, and dependency analysis are routinely used and software support resources are provided. An organization-widehistorical data base is used by managersfor project planning purposes.	Advancedplanning procedures and tools are consistently and regularly used for all projects within every phase of the life cycle, including all those mentioned before. Results are captured in an organization-widedata base and are used in conjunction with predictive methods. Planned and actual results are highly correlated to optimally plan projects.
Software Development Plans (Including Software Quality Assurance Plans) for Projects	There is no concern about formally planning software development projects. Any planningdone is informal (if done at all).	The importance of planning the software development projects starts to be recognized. However, only some projects within the organization actually write and ensure consistency of project plans. These efforts are decentralized and fragmented.	It is widely recognized that formal planning is an essential part of the software development process. Several projects follow written and consistent project plans However the plans are based on "best guesses" or the experience of the project managers	Many software development projects are formally planned and the plans created are consistent. Experience captured from earlier projects in the organization's historical data base is used, although this process is not very effective yet.	Almost all software development projects are formally planned and the plans are consistent. The selection of appropriate phases, deliverables, methods, approvals and tools to be used is done based on historical information readily available to the software manager	There are significant indications that all software development projects within the organization are formally and consistently planned based on data from the organization's historical data base. All projects produced within the past year have conformed to approved plans (up to now)
Selection of Effective Development Methods and Tools	There is no concern for selecting methods that are effective for software projects Software tools used in the organization often do not produce improvement.	Some project managers QUESTION 10.the suitability of the analysis, design and test methods traditionally used within the organization. Some projects are experimenting with alternative techniques and technology and/or tools	The importance of selecting and tailoring formalized methods is widely recognized. Data is collected about the successes and failures of methods currently used. Methodselection and tailoring is effectively done by some projects. Tools are often selected for specific reasons with projected impact.	The organization has specific recommendations for what formalized methods and technology and/or tools are suitable for which kind of applications. The recommendations are based on the historical data collected. Method and tool selection and tailoring are effectively done by several projects	There is evidence that almost all software managers select and tailor the formalized methods used in their projects Tools used by projects are effectively aligned with the methods they support and with each other. There are many positive results.	There are significant indications that formalized method and tool selection and tailoring have reduced problems of information representation and manual effort. Overhead reduction is obvious due to tool and methodtailoring.
Execution and Tracking of the Software Development Projects	There is no concern about tracking software development projects.	The importance of tracking software development projects starts to be recognized. Some projects use metrics for tracking purposes. However, these efforts are isolated and may not correspond to life cycle deliverables or activities.	Several projects use metrics to track their progress against their plan. Software managers get more involved in tracking life cycle deliverables and activities.	Many software development projects track their progress against their plan, as well as quality and productivity goals that they have set. Some of these projects enter this information in the organization's historical data base for use by future projects. Software managers for most of the projects are involved in tracking all life cycle deliverables and activities.	Almost all software development projects use metrics to track their progress and their managers are involved in tracking all activities and deliverables. These projects enter "lessons learned" into the organization's historical data base, which has been found to be useful by new projects.	There are significant indications that all software development projects within the organization formally track their progress and their managers actively track all activities and deliverables. Theorganization's historical data base is appropriately updated and it has been used effectively for planning purposes during the last year.

QUESTION 10.2 (cont.): Are software project planning and control mechanisms in place and followed?

THEME	POOR (0)	WEAK (2)	FAIR (4)	MARGINALLY QUALIFIED (6)	QUALIFIED (8)	OUTSTANDING (10)
Control Mechanisms (Milestones, Status Reviews, etc.) and Tools	No control mechanisms or tools are used. Their potential is not recognized.	Some control mechanism(s) and tools are under evaluation. Not widely used.	Several control mechanisms and tools are used within several projects by some of the staff. Historical data starts to be captured. Software Development Plans (SDP) are developed that are compatible with the Software Quality Assurance Plan (SQAP).	Common control mechanisms and tools are consistently used for most projects by most of the staff. SDPs (compatible with the SQAPs) are developed and used. Local historical data bases are established.	Common control mechanisms and tools are consistently used for all projects by the staff. All projects have an SDP (consistent with the SQAP) which they follow. An organizationwide historical data base is established and used by most projects for better project control.	Modern control mechanisms and tools (including SDPs consistent with SQAPs) are consistently and regularly used for all projects within every phase of the life cycle. Results are used (in the context of the historical data base) to optimally control projects.
Risk Management and Contingency Planning	No risk evaluation or risk management and contingency planning is performed on projects.	Some risks are analyzed and decisions made on them but no contingency plan exists. These efforts are isolated to some projects.	Most major risks are identified and acted upon, usually with adequate lead time. An informal contingency plan exists in some cases. Little or no historical data is kept.	Most risks are identified and acted upon, with adequate lead time. Contingency plans are formed for low priority or extremely unlikely risks. Usually positive results from them.	All key risks and many less likely risks are identified and acted upon with adequate lead time. Risks are minimized. When necessary, contingency plans are well executed.	All key risks, such as: schedule, conflicts, inadequate support, lack of follow through, etc., are understood, regularly reviewed and acted upon with adequate lead time (based on the project's contingency plan). Historical data are kept for projections and the development process is tailored to minimize or eliminate risks.

Appendix - D QSR and the SEI CMM

All audits and assessments have differences, advantages and shortcomings, so it is difficult to say whether the QSR Guidelines are better than the SEI Maturity Model, or vise versa. Since the QSR Guidelines and the SEI Maturity Model were originally developed in isolation of each other, the two models have some inherent differences. The people responsible for the software subsystem of the QSR Guidelines are constantly revising it and incorporating concepts from ISO-9001, SEI, ANSI/IEEE and other standards and models.

The following chart shows how the SEI Key Process Areas relate to the QSR software questions.

QSR #	Description	SEI Level KPA
10.1	Documented Process	2
10.2	Project Planning	2
10.3	Part of System, Entry, Exit, Reviews	2
10.4	Requirements	2
10.5	Configuration Management	2
10.6	Security and Archiving	
10.7	Independent System Testing	
10.8	Goals to Six Sigma Quality	4
10.9	SQA	2
10.10	Continuous Improvement	5
10.11	Capability and Training	3
10.12	Subcontractor Control	2

Differences between the QSR and SEI include:

- The QSR covers all aspects of the company including manufacturing, quality systems, and management as well as software. Software is audited within the framework of building products and running a business. The SEI Maturity Model focuses on software development only.

- The QSR has been institutionalized at Motorola. We are still struggling somewhat with SEI.

- All interviews in the QSR are one-on-one. SEI conducts group sessions where functional area representatives get a chance to talk candidly.

- The QSR gives an "analog" score between 0 and 100. SEI gives a "digital" score of 1 through 5.

- The QSR is very focused on documented evidence as well as interviews. SEI assessments tend to rely on project leader interviews and group discussions with less emphasis on documented evidence.

- The QSR Guidelines are "flat". The 12 software areas an organization needs to be qualified in are not prioritized. SEI acknowledges that an organization needs to be competent in basic key areas before working on more "mature" areas. Hence, the 5 level maturity model. The SEI CMM may allow an organization to focus better because of its 5 levels.

- The QSR auditors are all from a Motorola organization other than the one being audited. The focus is more of an independent audit. The SEI assessment team includes members from the organization being assessed. The term "assessment" implies the organization being assessed and the assessment team are working together with a goal of improvement. Note that the SEI assessment often appears as an audit to those being interviewed.

- An SEI Level 2 organization will rate a QSR software score of about 7 out of 10. But an organization rating a 7 QSR score may not be at SEI level 2. The discrepancy occurs because the QSR averages the scores of all software questions allowing high scores in some areas to make up for low scores in others. SEI requires a fully satisfactory level of competency in all key process areas.

- At Motorola, it is mandatory to have a QSR every two years. SEI scheduling is more or less up to the individual divisions. MDD is alternating the QSR and SEI assessment years so there is an audit/assessment every year, and a constant focus on improvement.

Since most of the basic principles of the QSR Software Quality Assurance subsystem and SEI Assessment are the same, MDD feels that the QSR has given us a leg up on SEI, and for all their differences, the QSR and SEI support each other.

INDUSTRY-WIDE SOFTWARE PROCESS IMPROVEMENTS

Wolfgang B. Strigel, M. Sc., MBA
Software Productivity Centre
Vancouver, Canada

Abstract:
The mandate of the Software Productivity Centre (SPC) of British Columbia is to further quality and productivity in the software development process of member firms. As a technical resource centre for software developers it provides practical services which target to increase effectiveness in dealing with common development issues.

During its first year of operation the SPC attracted over 75 industrial members. Workshops, seminars and courses were offered to share experience and to summarize new development practices. SEI-type assessments helped firms to pinpoint areas for improvement. Productivity was increased through standards compliant soft copy templates of most commonly used documents.

A survey of SPC members is currently conducted. The results will show which areas of support are most needed and how to streamline our program for the coming year. A summary of survey results will be presented at the conference.

Biography:
Mr. Strigel is the founder and Managing Director of the Software Productivity Centre. Previously he was Vice President of Engineering at MacDonald Dettwiler where he introduced a successful software engineering process using methodologies, standards, and metrics.

Mr. Strigel has over 20 years of experience in software development and management. He received a B.Sc. (1971, Munich, Germany) and M. Sc. (1974, McGill University, Montréal, Canada) in Computer Science and a MBA (1992, Simon Fraser University, Vancouver, Canada). Mr. Strigel is a member of ACM, senior member of IEEE. His particular interests are in software development methodologies, metrics, and process improvement. He is the author of several papers, and a member of IEEE and ISO standards working groups.

1. Introduction

Software development is no longer a problem. Many software companies have become famous for their growth rate and profits. Some have made spectacular gains on the stock markets. Software systems have become increasingly complex and their size is measured in millions of lines of code. During the eighties many large development projects gained an infamous reputation for huge cost overruns but recently the success stories are more abundant than horror stories. Have we really achieved the goal of moving from an art to the controlled engineering discipline of software engineering?

It is obvious that things have improved dramatically since Donald Knuth published his seminal work in seven volumes entitled "The Art of Computer Programming" [Knuth, 1972]. However we are entering a new phase in the evolution of software development and new challenges emerge. In his book "Decline and Fall of The American Programmer", Ed Yourdon leaves no doubt that the software industry is going to face enormous challenges. Software user expectations have quickly been raised to new levels. User friendliness, graphical user interfaces, and comprehensive help facilities are now expected. Time to market has become one of the key elements for competitive advantage. There are over 1500 CASE tools on the market. As soon as most of us become comfortable with structured methodologies and the waterfall life cycle, new techniques emerge which make those practices obsolete. Object oriented technologies are no longer a fad but will represent the dominant development technique by 1995 [Ovum, 1992].

Problems are compounded by rapidly shrinking profit margins. If sophisticated software packages as Microsoft's data base "Access" can be sold for under $100, there is significant price pressure on all software except for custom development and small niche markets. Competition is no longer limited to European or American firms. Offshore development allows the production of software for about $200 per function point instead of $1000 in North America [Yourdon, 1993]. Borland's "Quattro" was developed in Hungary at a fraction of the cost it would have commanded locally.

In this environment, efficient access to productivity enhancing technologies becomes a matter of survival. With the pace of product announcements quickening and with shrinking profit margins, companies are finding it increasingly difficult to spend time on so called "overhead activities." These include time spent on surveying the market for new technology developments, more efficient development tools, definition of internal procedures and methodologies, etc. A survey of 167 companies conducted by the Software Engineering Institute (SEI) showed that only 1% of respondents showed a process ranking of three or better on a scale from 1 to 5. The SEI assessment may not be the most appropriate measure and, as discussed below, it does raise some concerns.

In 1991 the Science Council of British Columbia conducted a similar survey of 30 B.C. software developers. None of the surveyed companies ranked better than SEI level 1. The survey also indicated that there was significant demand for an industry resource centre which could help companies to improve their effectiveness in software development. This result led to the creation

of the Software Productivity Centre of B.C. This paper describes activities of the SPC and its results after one year of operation.

2. The Software Productivity Centre (SPC)

The Science Council survey identified a need and a strong demand by B.C. software developers to establish an organization to support local firms. A business plan was submitted for provincial funding, and in 1992 the SPC was launched as a non profit organization. It is a member driven institution with the objectives to assist software developers in improving productivity and quality.

The concept of a technology transfer institution is timely and essential. According to Ed Yourdon, " technology transfer will be the biggest problem with new system development technologies in the year 2000, just as it was in 1990 and in 1980. You can read books, join research consortia, attend conferences,---. *But you have to get started now*" [Yourdon, 1992].

This is not a dramatically new insight for most IT professionals. Nevertheless it is not happening! In many cases there is no time, no budget, or no disaster yet, to raise the priority of technology transfer beyond the threshold of inertia. In many cases the SPC assumes the role of an evangelist, preacher or teacher. Companies don't perceive the need for process improvements until problems become so critical that development is severely impacted. We help our members to overhaul their development approach so that it can meet new needs as they arise. In many cases this has avoided costly mistakes which are more difficult to resolve if caught too late.

In a way the SPC implements the equivalent of the Japanese concept of a "keiretsu." The keiretsu is defined as "a confederation of loosely related companies based on having a common banker or supply network" [Rutherford, 1992]. Instead of a common banker we are a common source of technical expertise, an exchange of knowledge and experience that is shared among member companies. We see ourselves as the methodology department, the technology research department, or even the strategic planning department of our members. Not many professionals have the time to sit back and read predictions about future technology, or to ponder the impact of new developments on the longevity of their product. In other words, many firms are often too much in a hurry to stop for gas.

The following sections of this paper describe in more detail the services offered by the SPC to its industrial members.

3. SPC Services

3.1 Training

There are many good training programs in software related topics, and we don't see the need to duplicate any of the existing offerings. However, we did not find good courses on the process of software engineering. Early in 1992 we had the opportunity to review a training library on software engineering which was developed by the National Computing Centre (NCC) in the UK. This library consists of eight separate courses, each of which was designed for a 1 week intensive training course. The NCC SE training library has been offered world wide including countries such as Brazil, Singapore, Hongkong, etc. In England the course series is used in the context of training for IEE (Institute of Electrical Engineers, UK) certification of software engineers. We felt that the courses were well structured and addressed general software engineering topics not covered by locally available training programs. Therefore, the SPC decided to acquire the course library.

At this point we are offering 4 out of the eight available courses. Although the general course framework is excellent, we modified each course to delete some UK specific references and to adapt it to the North American context with respect to commonly used methodologies and CASE tools. We also added case studies to each course to increase its practical aspect. However, our members felt that a one week course is taking too much time from their busy schedules. We therefore compressed the courses to anywhere from 2 to 4 days. The courses are offered on Friday and Saturday during one or two consecutive weekends. This spreads the time commitment fairly between attendees and sponsoring companies.

We currently offer courses in Software Quality Assurance, Software Project Management, Principles of Software Engineering and Configuration Management. The courses are very popular, especially since they are taught by experts from industry who we hire for each course. These experts are familiar with software engineering problems from their daily work and can answer questions in a more practical way than professional instructors.

If a sufficient number of our members are interested in other courses such as language or UNIX training, we survey the market for the best commercial course provider and bring the course to B.C. This way our members don't have to travel to other cities to get this training and also benefit from a group discount which we are able to obtain.

3.2 Technology Transfer

Technologies surrounding software development are undergoing rapid change. As soon as one generation of computer chips has been introduced to the market, a new and more powerful set is already announced. Development methodologies are continuously improved and corresponding software tools are flooding the market. The only way a company can maintain its competitive edge is to constantly update its knowledge in emerging technologies. This is a difficult task given that most of us in the industry already are working well beyond the "mythical 40 hour week".

There are never enough people on a development team, and available development time seems to be always close to the pain threshold. Therefore it is not surprising that activities related to maintaining awareness of new technologies are typically short changed. It is in this area where the SPC can offer real practical help.

As in all of our activities, we are driven by the needs of our members. Through a variety of feedback mechanisms we keep in touch with the needs of our membership. Based on their needs we survey the literature, attend conferences or communicate world wide through Internet to find the best sources of pertinent information. Rather than passing on a wealth of unqualified information we filter out the essential and pass it on through presentations, seminars, workshops etc.

Sometimes we organize seminars for a large audience and invite leading experts or "gurus" in the respective field. But we found that in many cases we don't need to invite the proverbial guru from far away. Much expertise is resident in local companies which had to specialize in some area or who have gained experience through trial and fault. We can help to spread this local experience to the benefit of other companies.

To facilitate exchange of local experience we organize workshops or special interest groups. Members of each group meet periodically to exchange their experience or to work together on exploring new technologies. In the latter case the SPC can assist by researching a topic through literature searches, obtaining conference proceedings or even by acquiring relevant market reports.

In all cases we found that our success is based on our practical focus. We emphasize proven industry experience over theoretical lectures. Most members ask us to find practical "how-to" knowledge or solutions that can be readily applied to their problems. Obviously there is a place for leading edge research but our experience has shown that most of our member companies want practical solutions to their immediate problems.

3.3 The SPC Zoo

Practically every issue of most technical journals has articles or advertisements about new software tools. We are often flooded by new tools which promise to solve everyday problems in software engineering. However, we also see daily evidence that many of these tools end up as "shelfware". There are many reasons for this shelfware phenomenon. In some cases it is simply based on insufficient information. But more commonly there are two specific reasons: First, the organization is not ready to use the tool. Their current software engineering practices are not in line with the underlying paradigm of the tool. Simply installing the tool on an engineer's workstation will not change his fundamental approach to development. Secondly, management too often assumes that bright engineers should be able to learn the use of a tool by osmosis and training is therefore an unnecessary expense.

The cost of an unused tool is typically not the biggest financial risk. A much higher waste is observed if a company decides to change their process to suit a new development tool and finds out a year later that this approach is unsuitable for their operation. In this case endless hours are wasted to introduce the new approach and an equal amount of time can be spent to correct the fatal decision.

It became obvious that a facility to showcase a broad spectrum of tools would be useful for our members. We approached several platform vendors and started to establish a computer lab, initially with three UNIX machines. The term "computer zoo" was coined to describe the lab (the animals being the workstations). The zoo rapidly became very popular among our members and in early 1993 we expanded by adding three PC platforms. The SPC zoo now contains over 60 software tools from about 40 sponsor firms running under UNIX and Windows. We offer our members advice about tools and their applicability to their particular environment and needs. This happens without any vendor pressure, and software engineers can try out a variety of tools before making a buying decision. A database of 1150 tools assists us in choosing the tools to install in the zoo. Only high quality tools are installed, and, when in doubt, we check user references before installing a new tool.

3.4 SPC Assessment

Several years ago the Software Engineering Institute at Carnegy Mellon University (SEI) was tasked by the Department of Defense to develop an assessment tool which determines the capability of an organization to develop software in a reliable fashion. The resulting Capability Maturity Model (CMM) has since gained widespread popularity and has become a new standard which identifies on a company wide basis 5 levels of maturity within the software development process (levels 1 through 5). Within this model, every level is assigned a set of key practices, each of which must be implemented in order for a company to achieve that level. The levels represent increasing degrees of maturity within the software development process and a company must achieve each lower level completely before being qualified for the next higher one. The premise of the SEI model is that the higher the maturity level, the greater the company's ability to produce high quality software productively [Humphrey,1990].

Despite its popularity, the SEI assessment has some weaknesses. Its most significant shortcoming is that it assesses a company without regard for company size, business environment and company objectives. It was developed for large aerospace and defense contractors and uses terminology which is unfamiliar to smaller companies or MIS departments. The ranking scheme would not be very constructive for our purposes, as most B.C. based companies would not achieve any level higher than one.

After participating in a formal SEI assessment of a medium size telecommunication firm we decided to modify the assessment to suit our local industry. Our key objective was not to derive any particular ranking but to obtain recommendations with are directly applicable to improve the development process. We maintained most of the CMM concept and introduced the following changes.

First, we made the terminology understandable for small and medium size companies. Then we reduced the size of the questionnaire by eliminating questions which are only relevant for very advanced firms. It is interesting to note that this did not result in deleting all level 5 questions but rather a selection of questions from level 3 to level 5. Most importantly, we added the concept of considering the particular characteristics of the assessed firm. This includes business objectives such as growth targets, single or multiple product lines, product life expectancy, time-to market goals, etc. We also distinguish between shrink-wrap product companies, system integrators or internal MIS departments. It is inconceivable that companies with such diverse application environments should have to meet exactly the same evaluation criteria as the SEI model suggest. The following diagram shows how this additional information feeds into the original SEI process of team interviews, and assessment of findings.

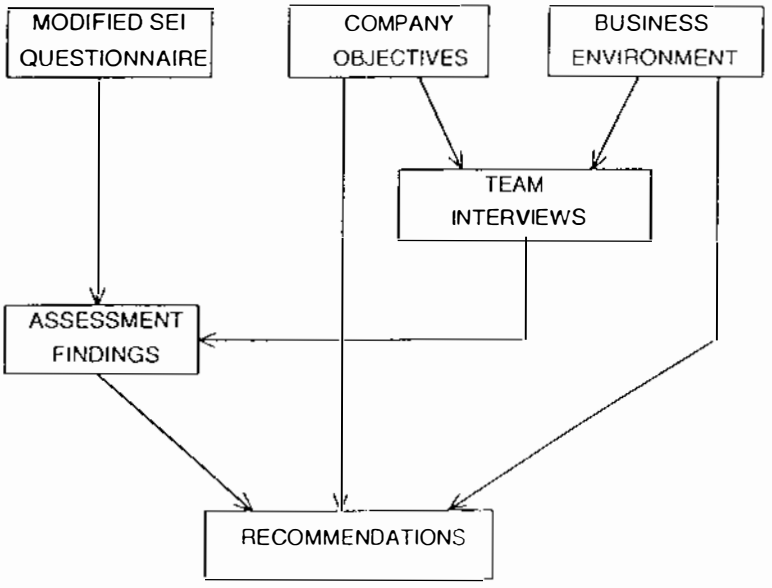


Figure 1: SPC Assessment Process

The SPC self-assessment process is designed to be used by a company to assess its own software process with some guidance from the SPC. It requires minimal company staff time to conduct, and can normally be completed within 3 calendar weeks (dependent on the size of the organization). Recommendations are based on the deficiencies found in key practice areas and on the company's business objectives and software development priorities.

3.5 Documentation Templates

For whatever reason it seems that reinventing documentation standards at the beginning of a new software project is a favorite activity. Specification, design and test documents are an essential part of each software development project. The contents of these documents are specific to each project. However, there is no reason why the general layout of the documents cannot be the same. There are several documentation standards which give guidelines for the layout. Nevertheless our observation has been that project leaders tend to redefine the layouts for each new project. The result is that documentation is not uniform within one company and if parts of the software are reused in subsequent projects the related documentation elements are not compatible. It also results in wasted effort and decreased productivity.

Documenting software development activities is not a difficult task in itself. But it is often neglected and in many cases documentation becomes an afterthought rather than an integral part of the development process. We believe that making the task of documenting easier can have a very beneficial impact on the overall development. In early 1993 we surveyed the market to find out if there were any user-friendly documentation templates in soft copy format. We sent out an inquiry to some 250 experts world wide through Internet. To our surprise we did not find any. The only replies we got asked us to send a copy if we were to develop something like that. This encouraged us to develop our own set of soft copy documentation templates.

The objective was to generate templates which are in line with existing documentation standards by IEEE and simultaneously comply with requirements spelled out in ISO 9000-3. An industry steering committee was convened with representation from small and large companies across a broad spectrum of applications. During the following six months we developed templates for 4 documents: Initial Requirements Specifications, Detail Requirements Specifications, Quality Plan, Development Plan. Each template consists of a table of contents, followed by generic sections. Each section contains a guide which describes in several paragraphs the type of information which should be contained in the section. The templates are accompanied by a user manual which gives a detailed description of how to use the templates and how to handle version control. These templates assist and simplify the documentation task. A software engineer just loads the template into his word processor, reads the guide for a particular section and then enters project specific material.

At the time of this writing (June 1993) the templates are just being launched and there is already a keen interest among our members. We believe that the templates help to increase productivity since no time has to be spent deciding on format and contents. But more importantly they make it easy for project staff to document their work throughout development. The resulting documents are consistent from project to project and comply with recognized standards. Companies can show these documents to customers or business partners as proof of a well managed and controlled development process.

4. Conclusion

After one year of operation the SPC is encouraged by its initial success. Membership went from zero to 75 corporate members and 36 sponsors. We organized about 15 courses and seminars and have three special interest groups. The computer zoo enjoys increasing popularity and has grown to one of Canada's largest showcases of software tools. We believe that this success proves the need for a resource centre in B.C.'s software industry.

We see the SPC not as an industry association but as a business. Our customers are our members and user satisfaction and relevance of services are just as important than in any other business. We are still partly subsidized by the provincial government but our goal is to become self sufficient. This should be feasible as our services continue to be of real value to the industry.

As we are heading into our second year new challenges are waiting for us. The renewal rate of our first year members will be the ultimate proof of our success. Our staff has grown to 5 full time employees and we are planning to add another software engineer in the fall. Much of our success to date is based on our clear focus on practical technical issues. This gives us a clear distinction from industry associations. Many of our members would like us to provide other services such as marketing assistance. A member survey is currently being conducted to help in establishing our second year goals. Once the results are available we will adjust our program accordingly.

Bibliography

Humphrey, Watts; "Managing the Software Process", Software Engineering Institute, Addison Wesley, 1990.

Ovum Report, "Object Technology", Judith Jeffcoate, Allison Templeton, Laurent Lachal, Ovum, 1992

Rutherford, Donald; Dictionary of Economics, Routledge, 1992

Yourdon, Edward; "Decline and Fall of the American Programmer", Yourdon Press, Prentice Hall, 1992

Yourdon, 1993: Ed Yourdon presentation to CIPS, Vancouver, Canada.

Brain Children Transform Chaos Manor into Productivity Palace

Sarah L. Sullivan, Ph.D.
Assistant Professor
Department of Computer Science
Indiana University - Purdue University
Fort Wayne, IN 46805

Abstract
With the help of the Brain Children a low quality and low productivity culture is transformed into a high quality and high productivity culture. The Brain Children embody the concepts of Continuous Process Improvement, Keeping It (the process) Simple, Collaborative Teamwork, and Technology Transfer. As a memory aid, these concepts are presented in story format.

Sarah L. Sullivan, Ph.D. received the B.S. in computer science from Bowling Green University (Ohio) in 1975. She received the M.S. and Ph.D. in computer science from Illinois Institute of Technology in 1986 and 1990 respectively. Dr. Sullivan's industry positions have ranged from programmer to second level software manager to management consultant. Upon completing the doctorate, she made a career transition into academia where she devises innovative teaching techniques to convey computer science concepts. Dr. Sullivan approaches quality and productivity by focusing on the human side of the software development process.

Scholarship donations are requested to fund student research on women in science. Please make check payable to: Indiana-Purdue Foundation at Fort Wayne. Please designate that the donation is for: "Women's Studies Scholarship Fund for Research on Women in Science." Please send it to: IPFW Development Office, 2101 East Coliseum Blvd., Fort Wayne, IN 46805-1499.

Chaos Manor

Once upon a time there was a little company named Software House that was more commonly referred to as Chaos Manor. Everyone at Chaos Manor was very smart, very very industrious, and very very very frustrated. They were frustrated because productivity and product quality were at unacceptable levels and everyone knew it. Attempting to turn things around, everyone worked harder and harder. But this made the chaos worse and worse. As the chaos got worse and worse, the productivity and product quality got lower and lower, and everyone got more and more frustrated. [1]

LeftBrain Encounters RightBrain

One lunch break, while strolling through the forest behind Chaos Manor, Techi LeftBrain collided with Genie RightBrain. After recovering from the embarrassment of having previously been oblivious of each other, they formally introduced themselves. As they talked, they discovered that their respective departments had been working at cross purposes. So, they decided to put their heads together. [2]

The Brain Children

This very fertile union produced quadruplets. RightBrain and LeftBrain named their Brain Children: Evolvo, KISS, CoLabor, and Demo.

Instead of hearing baby talk, the Brain Children grew up listening to their parents engage in problem solving discussions on work related issues. Much to everyone's surprise, as soon as the quadruplets could talk, they started participating in these discussions too.

Even more surprising were the personalities that emerged: Evolvo became a staunch advocate for evolving the software process through continuous process improvement; KISS delighted in finding elegantly simple ways to do or say things; CoLabor excelled in the subtle interpersonal techniques that foster collaborative teamwork; and Demo seemed to always be the first one to try to do something and therefore would always be the first to be able to give a validated example of how to go about doing it.

The Transformation

Everyone at Chaos Manor recognized that the Brain Children epitomized the very essence of what Chaos Manor needed to turn itself around. The Brain Children recognized it too.

With the help of CoLabor, Perky Personella and Labyrinth Legalese came up with a scheme to legally hire the Brain Children. Personella and Legalese argued that since there were no other known individuals whose right brain and left brain were fully integrated, the Brain Children were a new more advanced species and therefore the child labor restrictions did not apply. Furthermore, since the Brain Children were different from everyone else, to deny them employment would violate equal opportunity legislation. The argument was accepted and the Brain Children were hired. [3]

On the Brain Children's first day of remunerative employment, everyone gathered in the auditorium where Systema Software, the company founder, announced that the Brain Children would serve as company wide consultants and would be available to help anyone with anything. [4]

From that day forward the chaos progressively diminished and the productivity and product quality progressively improved. Soon everyone started referring to Software House as Productivity Palace. As this new reputation became more and more widespread, people started coming from far and wide to find out what Software House was doing that was working so well.

One day, Relay Reporta from the Quality and Productivity Gazette interviewed the Brain Children. Reporta wrote the following feature articles:

Champion [5] **of Process Evolution** [6]

Evolvo BrainChild's mission at Software House is to debug the system for building systems.

When Evolvo arrived, there wasn't any system for building systems. Instead, everyone was driven from crisis to crisis by unplanned priorities and unmanaged change.

Evolvo's initial system for building systems included the development of a project plan along with performance tracking and change control procedures.

Evolvo has championed the continuous improvement of this initial system. This evolution has been aided by the use of the software inspection method [7], walkthroughs [8], direct observation [9], formal and informal interviews [10], and project post mortems [11].

Through Evolvo's continuous process improvement efforts, Software House became the first company in the world to achieve level five of the Software Engineering Institute's Process Maturity Model [12].

Champion of KISS [13]

KISS BrainChild's mission at Software House is to eliminate unnecessary complexity.

When KISS arrived, almost all written communication had a fog index [14] above the doctoral level. This contributed to a readers and writers problem [15]. People who were writing did not have time to read. Often they did not realize that what they were writing was already written somewhere. People who were reading had to work too hard at seeing through the fog. They rarely got anything else done. People who were working did not have time to read or write. They never seemed to have the information they needed and their results rarely got written down.

KISS eliminated these communication breakdowns: by championing the use of software engineering structuring techniques [16], by requiring that all project documents be at or below the eighth grade reading level [14], and by challenging people to contain the articulation of a single thought to a single page [17].

Champion of Collaborative Teamwork [18]

CoLabor BrainChild's mission at Software House is to foster collaborative teamwork.

When Colabor arrived, the environment was so competitive that no one would help anyone with anything.

CoLabor has championed changes that have produced a truly team-of-all-equals egoless

culture [19]. This culture nurtures continuous improvement of the interpersonal effectiveness skills necessary for collaborative teamwork [20]. Additionally, the restructured reward system [21] encourages collaboration.

CoLabor's efforts have been so successful that counterproductive dominance jockeying and dirty politics have been eradicated.

Champion of Technology Transfer [22]

Demo BrainChild's mission at Software House is to encourage people to learn new more productive ways of doing things.

When Demo arrived, everyone was "gun-shy" about using new technology. Projects that used a technology for the first time tended to be dismal failures. These failures were typically followed by a mass firing of project personnel.

Demo promoted a four-phase approach to technology transfer. Demo introduced this approach in a manner that produced an example of all four phases.

With Demo's help, everyone has learned how to develop the examples needed for successful technology transfer. This has facilitated Software House's ability to explore and evaluate new ways of doing things. The resulting expertise has helped Software House to gain and maintain competitive advantage.

Productivity Palace

The exemplary corporate culture at Software House fosters peak performance [23]. With the help of Evolvo, KISS, CoLabor, and Demo, Software House has created a climate where people work smarter.

Software House's defect free product quality is the highest in the industry. Before any product leaves the premises, external auditors certify that all requirements have been meet. There have been no defect reports on any delivered software or on any accompanying deliverables since this step was added to the software process five years ago.

The defect free product quality has contributed to productivity levels that are twice as high as any other software development firm in the world. Productivity is so high because rework is so low.

In addition, Software House has been ranked highest in the industry for quality of work life.

Thanks to their outstanding productivity, product quality, and quality of work life, they all lived happily ever after.

FOOT NOTES:

[1] In an excessively stressful work environment, things tend to get worse and worse when people try harder and harder. How can this be? Software development is a complex intellectual process involving precise communication of complex abstract concepts across

multiple discipline boundaries. Excessive stress (distress) reduces the ability to deal with complexity. It also reduces the ability to communicate effectively. Managing software project stress is key to improving quality and productivity.

Sullivan, S. L. and Hill, H. (1987). "Software Project Stress versus Quality and Productivity." Proceedings of the National Computer Conference, 56, pp. 199-203.

[2] Much of the work of developing a software system involves: verbal communication, dealing with abstractions, and conquering complexity by organizing it into step by step logical sequences. These are the specialties of the left hemisphere of the brain [Blakeslee 1980].

Also, much of the work of developing a software system involves: non-verbal communication, creativity and intuition, and a holistic "feel" for a system. These are the specialties of the right hemisphere of the brain [Blakeslee 1980].

A synergistic working partnership between the right and left brains needs to be cultivated and nurtured. The Brain Children symbolize four instances of right-left synergy.

Blakeslee, T. R. (1980). The Right Brain: A New Understanding of the Unconscious Mind and Its Creative Powers. New York: Berkley Books.

[3] This process of coming up with a scheme to legally hire the Brain Children is an example of collaborative teamwork. (See [18])

[4] Through this meeting, top management demonstrated a commitment to improving quality and productivity through: continuous process improvement, keeping it (the process) simple, collaborative teamwork, and technology transfer. Additionally, top management publicly empowered all employees to use these tools.

[5] Changes such as those epitomized by the Brain Children require a strong advocate. This advocate is often referred to as a champion because of the endurance required to "hang in there" until the change is firmly in place.

[6] Software systems are built in a rapidly changing environment. These changes include new or improved computer hardware, peripheral devices, operating systems, manual and automated software development tools and methods. While such changes are intended to improve the system for building systems, this is not necessarily the case. A champion of process evolution is needed to guard against incorporating counterproductive changes into the system for building systems.

[7] An inspection is a formal proof reading process. Checklists are used to ensure that the inspection procedures are followed. Statistics are kept on the number of defects (errors) detected and on the defect detection rate (errors found per inspection hour). The inspection moderator records the defects found and later follows up to ensure that these defects got corrected.

Brykczynski, B. and Wheeler, D. A. (1993). "An Annotated Bibliography on Software Inspections." Software Engineering Notes, 18(1), pp. 81-88.

[8] A walkthrough is a peer group review of any work product. The purpose is to provide constructive feedback to the person who produced the work product so that (s)he can improve it. This feedback is kept at a peer level. It is neither passed along to management nor to other groups.

Yourdon, E. (1989). Structured Walkthroughs 4th ed., Prentice Hall.

[9] Direct observation refers to directly experiencing a situation using one's own senses. It involves the right brain. Direct observation allows one to capture aspects of a situation that are difficult to accurately convey verbally.

Weinberg, G. M. (1988). "Observation." Chapter 3 in Rethinking Systems Analysis & Design, Dorset House Publishing, pp. 43-60.

[10] Formal and informal interviewing require identifying the right questions to ask and phrasing questions such that the responses indicate whether the questions were understood.

Weinberg, G. M. (1988). "Interviewing." Chapter 4 in Rethinking Systems Analysis & Design, Dorset House Publishing, pp. 61-86.

[11] A project post mortem is a post project review. Its purpose is to assess what went well (successes) and what could be improved (hindsights and insights). These results are used in evolving the software process.

Whitten, N. (1990). "Post-Project Review: Understanding the Past to Improve the Future." Chapter 12 in Managing Software Development Projects, Wiley, pp. 241-258.

[12] The five levels of process maturity are:

1. Initial: No formal procedures, estimates, or plans exist.
2. Repeatable: Reasonable control of schedules. Informal and ad hoc process methods and procedures exist.
3. Defined: The organization has achieved the foundation for major and continuing progress.
4. Managed: Data is gathered and used to improve the quality of the product.
5. Optimizing: Data is gathered and used to tune the process itself. Emphasis is on tuning the software development process to prevent the introduction of defects (errors).

The following reference discusses each maturity level and the actions required to advance from one maturity level to the next.

Humphrey, W. S. (1989). Managing the Software Process. Addison-Wesley.

[13] Keep It Simple, Smartie! It takes a lot of smarts to simply and clearly express a complex abstract concept in such a way that it can be readily and accurately understood by others.

Because of the prevalence of communication breakdowns on software projects [Krasner et al. 1987], the most valuable project team members [Guinan and Bostrom 1987] are those who are proficient in translating information into a form that can be received effectively by others.

Guinan, P. and Bostrom, R. (1987). Communication Behaviors of Highly-Rated vs. Lowly-Rated System Developers: A Field Experiment (IRMIS Working Paper #W707). Bloomington, IN: Indiana U., School of Business.

Krasner, H.; Curtis, B.; and Iscoe, N. (1987). "Communication Breakdowns and Boundary Spanning Activities on Large Programming Projects." Proceedings of the 2nd Workshop on Empirical Studies of Programmers. Ablex. pp. 47-64.

[14] Gunning's Fog index enables writers to determine how many years of schooling are required to read a given document with ease. To compute the fog index, apply the following formula to a sample of at least 100 words:

GradeReadingLevel = 0.4 * (AverageWordsPerSentence + PercentWordsMoreThan2Syllables)

Ten principles to prevent written fog [Mueller] are:

* Keep sentences short.
* Prefer the simple to the complex.
* Prefer the familiar word.
* Avoid words you don't need.
* Put action into your verbs.
* Use terms your reader can picture.
* Tie in with your reader's experience.
* Write the way you talk.
* Make full use of variety.
* Write to EXpress, not to IMpress.

Mueller, D. (1980). "Put Clarity in Your Writing." IEEE Transactions on Professional Communication, PC-23(4), pp. 173-178. (Reprinted in: Writing & Speaking in the Technology Professions: A Practical Guide. Edited by D. F. Beer, IEEE Press, pp. 25-30.)

[15] The four types of communication breakdowns identified by Krasner [1987] are:

* no communication between groups that should be communicating,
* miscommunication between groups,
* groups receiving conflicting information from multiple sources, and
* communication problems due to project dynamics (i.e. As the project evolves, project information changes [Floyd 1988]).

Floyd, C. (1988). "Outline of a Paradigm Change in Software Engineering." Software Engineering Notes, 13(2), pp. 25-38.

Krasner et. al. (1987) See [13] above.

[16] Numerous software engineering structuring techniques [Martin and McClure 1988] have been developed to aid the non-ambiguous articulation of complex abstract mental models. The software engineering sub-discipline of computer science has matured to the point where a master of science program devoted to the study of software engineering is available at selected universities [Ardis and Ford 1989].

Ardis, M. and Ford, G. (1989). 1989 SEI Report on Graduate Software Engineering Education. Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, Technical Report CMU/SEI-89-TR-21.

Martin, J. and McClure, C. (1988). Structured Techniques: The Basis for CASE. Prentice Hall.

[17] The one-page rule is:

> If it fits on one page,
> it is more likely to get read.

One-page documents can readily: be posted, written on and forwarded, attached to a cover memo, FAXed, or quickly transformed into electronic mail and distributed. Adherence to the one-page rule increases the chances that information will actually get to the personnel who need that information to do their piece(s) of the project.

[18] A team is working together collaboratively when: 1) they synergistically pool their collective talents and resources as they work jointly toward a common goal, and 2) the outcome is greater than the sum of the individual contributions, and 3) every member of the team wins.

[19] The team-of-all-equals structure of the egoless programming team model [Weinberg 1971], has been found to produce superior results compared to a hierarchical structure [Krasner et. al. 1987, p. 59]. Research on gender communication differences [Tannen 1990] has revealed that this team-of-all-equals structure, while familiar to women, is typically foreign to most men. Management techniques that encourage a team-of-all-equals structure in both all-male and mixed gender teams are currently being developed [Sullivan 1993a and 1993b].

Krasner et. al. (1987) See [13] above.

Sullivan, S. L. (1993a). "teams of equals engender change" Current: The IPFW Quarterly Magazine, March, pp. 12-13. Fort Wayne, IN: News Bureau and Publications, Indiana University - Purdue University.

Sullivan, S. L. (1993b). "Engendering an Egoless Culture." Working Paper.

Tannen, D. (1990). You Just Don't Understand: Women and Men in Conversation. Morrow.

Weinberg, G. M. (1971). The Psychology of Computer Programming. Van Nostrand Reinhold Company.

[20] Interpersonal effectiveness skills necessary for collaborative teamwork [Sullivan 1990] include: active listening [Rogers 1957], appropriate use of talk styles [Miller et. al. 1988], communication awareness [Miller et. al. 1988], nonverbal communication [Mehrabian 1981], assertiveness [20a], win-win problem solving and conflict resolution [Gordon 1977], participative leadership, and an understanding of group dynamics [DeMarco and Lister 1987].

[20a] Assertive behaviors generally strive toward equality and symmetrical exchanges.

Demarco, T. and Lister, T. (1987). Peopleware: Productive Projects and Teams. New York, NY: Dorset.

Gordon, T. (1977). Leader Effectiveness Training. Bantam Books.

Mehrabian, A. (1981). Silent Messages. Wadsworth Publishing Co.

Miller, S.; Wackman, D.; Nunnally, E.; and Miller, P. (1988). Connecting with self and others. Littleton, CO: Interpersonal Communication Programs, Inc.

Rogers, C. (1957). Active Listening. Chicago, IL: University of Chicago Press.

Sullivan, S. L. (1990). Assessing the Effect of an Instructional Intervention on the Communication Behaviors of Software Engineers. (Doctoral dissertation, Illinois Institute of Technology, University Microfilms International 9030480)

[21] That which gets fed, grows. That which does not, withers and dies.

Teamwork rewards must be given equally to all members of the team being rewarded (Otherwise, they are individual performance rewards).

Ideally, rewards for outstanding collaborative teamwork are given publicly with much hoopla.

[22] Technology transfer refers to the assimilation of new technology by an organization.

Learning the technology well enough to produce an example of its use is the first step of this assimilation process. This experimentation should produce a throw away project or "toy" system.

The second assimilation step involves employing the technology on (in) a pilot project. This project should stimulate insight into the potential practical applications of the technology. It should also stimulate insight into the impact of the technology on the software development process.

Once the implications and applications of the technology are understood, an example methodology can be devised.

These first three steps involve the development of a lot of examples.

After the example methodology has been "debugged," the groundwork has been laid for the fourth step which is widespread use of the technology by the organization.

Cash et al. [1992] present these four steps as necessary for technology transfer to succeed.

Cash, J. I.; McFarlan, F. W.; McKenney, J. L.; and Applegate, L. M. (1992). Corporate Information Systems Management: Text and Cases, 3rd ed., Irwin.

[23] The relationship between stress and performance is an inverted U-shaped curve [Ivancevich and Matteson 1980] with peak performance occurring at the optimum stress point.

Garfield [1986] found that peak performers are motivated by a mission, they possess the twin capabilities of self-management and team mastery, and they have the ability to correct course and manage change. Peak performers use these attributes to achieve and maintain optimum stress.

Optimum stress maximizes communication effectiveness and the ability to deal with complexity. Hence, it creates the highest potential for quality and productivity [Sullivan and Hill 1987].

Garfield, C. (1986). Peak Performers: The New Heroes of American Business. New York: William Morrow and Company.

Ivancevich, J. M. and Matteson, M. T. (1980). "Stress and Performance." in R. M. Steers and L. W. Porter (eds.), Motivation & Work Behavior, 3rd ed., McGraw-Hill.

Sullivan and Hill (1987) See [1].

Risk Management and Quality in Software Development

Ronald P. Higuera and David P. Gluch
Software Engineering Institute[1]
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213
rph@sei.cmu.edu
dpg@sei.cmu.edu

Abstract: This paper discusses the role of risk management and quality in the development of software-dependent products. Specifically, the importance of risk management in meeting the quality goals of developing the right product and developing the product right within an ever decreasing time-to-market is addressed. The elements of a comprehensive Team Risk Management (TRM) methodology, including the central process elements of identification, analysis, planning, tracking and control, as applied to software technical risk management are presented. In addition, an overview of the preliminary observations of recent implementations of the techniques and a summary of related lessons learned are discussed.

Biographical: Ronald P. Higuera is a Senior Member of the Technical Staff and project leader at the Software Engineering Institute (SEI). He first joined the SEI in 1988 with the Process Program working to enhance software acquisition through process improvement. His Team Risk Management Project is an integrated product team approach to promote cooperative risk management methods between the government program offices and their industry partners. Before joining the SEI, he spent 20 years in weapon system acquisition in the US Air Force, most recently at the Defense Systems Management College (DSMC) where he was Director, Software Acquisition. He is a Senior Member of the IEEE and a member of the Association of Computing Machinery. He is the co-author of the DSMC textbook, *Mission Critical Computer Resources Management Guidebook*, published by DSMC in 1988. He holds a master's degree in electrical engineering and is a graduate of the Program Management Department at the Defense Systems Management College.

David P. Gluch is a Senior Member of the Technical Staff with the Software Engineering Institute at Carnegie Mellon University. He has been involved with the development of real-time computer and software-intensive systems including real-time UNIX and embedded systems for more than fourteen years. In addition to engineering and management responsibility for major projects, he has co-authored a book on real-time UNIX systems, authored numerous corporate technical reports, and has published more than 20 articles on topics ranging from general purpose real-time computers to fly-by-wire flight control systems. Prior to his work in industrial R&D he held various academic research and teaching positions. He holds a PhD in physics from Florida State University and is a Senior Member of the IEEE.

---

[1] The Software Engineering Institute is sponsored by the U.S. Department of Defense.

Introduction

In today's economic environment the race for quality is the race for survival and the mark of quality has emerged as the winning edge required to realize success. But further complicating the challenges of competition in today's economy is the fact that the economic environment itself is changing at an ever-increasing rate. This increasing rate of change has resulted in a shortened time-to-market [Vesey 92], [Akao 90] and has required management to be increasingly anticipatory in their decision making processes. Recognizing uncertainty, anticipating potential adverse consequences and initiating proactive management practices leads to fewer problems, fewer crises and greater success throughout the life-cycle of a product. This anticipatory characteristic is a key element of effective risk management and ultimately of product quality.

Quality as discussed by Juran has a dual meaning: product features that meet customer needs and freedom from deficiencies [Juran 89]. Building the right product and building the product right are necessary and complementary factors toward customer satisfaction. Meeting time-to-market is equally important in achieving customer satisfaction and is critical to an enterprise's survival. Risk management provides a means to achieve all three: building the "right" product, building the product "right," and delivering the product at the "right" time.

Fundamentally, risk management is an informed decision making approach that involves consciously anticipating what could go wrong, assessing the potential loss (severity of the impact) and incorporating this broadened perspective into program planning, program activities and program decision making processes.

Software Risk Management

The Software Engineering Institute's (SEI) paradigm for managing risk, shown graphically in Figure 1 [SEI 92], draws many parallels with Juran's three "universal" processes for managing for quality: quality planning, quality control, and quality improvement [Juran 89]. A concurrent strategy of quality management and risk management provides a comprehensive foundation for realizing success in software-dependent development programs.

The elements of the SEI risk management paradigm are [SEI 92]:

1. **Identify** – search for and locate risks before they become problems and adversely affect the program.
2. **Analyze** – process risk data into decision making information.
3. **Plan** – translate the risk information into decisions and actions (both present and future) and implement these actions.
4. **Track** – monitor the risk indicators and known risks.
5. **Control** – correct for deviations from the planned risk actions.
6. **Communicate** – provide feedforward and feedback data internal and external to the program on the risk activities, current risks, and emerging risks.

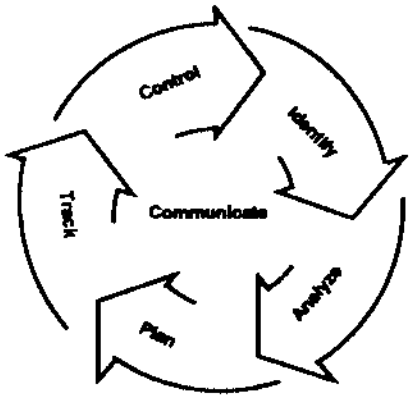


Figure 1. Risk Management Paradigm

The paradigm is represented as a wheel to reflect the fact that risk management is a continuous process. We live in a dynamic world and risks continually change and new risks emerge. Effective risk management requires a constant vigilance relative to customer satisfaction and the changing environment.

At the center of the paradigm is communication. To effectively identify, analyze and manage risks, information relating to risks must be communicated freely at all levels throughout the organization and between organizations. This broad organization-wide communication characterizes a "risk aware" culture [SEI 92] that views knowledge of risks as vital to successful program management.

Team Concepts

Another factor that is key to successful product development and quality is a "team" approach. Other terms have been used to describe this process of working collectively toward common goals, such as concurrent engineering, which involves the integration of product and process design [Winner 88]. This team approach is also a vital strategy for meeting time to market goals and is a key element in risk management.

The "team" characteristic of quality is evident in the approach of Deming in Point 4 of his 14 Points for Management. Specifically,

> "4. End the practice of awarding business on the basis of price tag. Instead, minimize total cost. Move toward a single supplier for any one item, on a long-term relationship of loyalty and trust." [Deming 82]

J. M. Juran regards the teamwork relationship as continuous and planned, between supplier and buyer, and as a relationship in which both parties "work together as if they were

part of the same company." [Juran 88] This teamwork is based upon:

- mutual confidence
- joint planning
- mutual visits
- mutual assistance
- no secrets

Throughout the foundations and implementation of quality practices, the concept of teamwork is integral to the effective management and realization of quality in business endeavors.

As applied to software development, a team-oriented organization not only addresses meeting the schedule, cost, and performance objectives of a program by addressing the technical and programmatic managerial aspects but is also concerned with issues relating to interpersonal relationships and communication [Kezsbom 89]. The communication that characterizes these interpersonal relationships is a key element in the implementation of team risk management within an organization and between organizations in a government-contractor or contractor-subcontractor relationship.

Fundamentally, the team risk management concept is an extension of the working team (single supplier) quality concept to include, not only the quality of the products or services delivered, but also the management of the product development process itself. Specifically, the implementation of team risk management, between government and contractors or contractor and subcontractor, is the application of the concept of teamwork in buyer-supplier relationships to the management of the uncertainties (risks) in the product development process.

The concept of team risk management is an amalgamation of risk management methods, quality-focused methods, and participatory (team-oriented) management concepts. The genesis of the approach is based in the work of a diverse group of technical, management and quality-oriented professionals, and is the culmination of collaborative research, development and testing in the area of risk management between the SEI and its government and industry clients[1]. Fundamentally, the team risk management methods and tools developed by the SEI represent a pragmatic application of these principles.

Team Risk Management

An integrated team approach to risk management, founded in effective communications, is one of the important aspects of the SEI team risk management approach [SEI 93]. Communication enables the dynamics and synergy that characterize effective risk management and result in a collective "team" insight and overall effectiveness that is substantially greater than the sum of the separate contributions of each individual.

---

[1] The Risk Management Program at the SEI was initiated in January of 1990 and has benefited from the contributions of numerous client organizations, affiliate members and SEI professionals.

Team risk management is founded upon the nine principles summarized below [SEI 93]:

The Principles of Team Risk Management

1. **Shared product vision**
2. **Forward-looking search for uncertainties**
3. **Open communications**
4. **Value of individual perception**
5. **Systems perspective**
6. **Integration into program management**
7. **Proactive strategies**
8. **Systematic and Adaptable methodology**
9. **Routine and continuous processes**

Collectively, the principles identified above form the basis for the processes, methods and tools that embody the concept of team risk management. The methods and tools of team risk management as they relate to the basic Software Engineering Institute's Risk Management Paradigm are represented graphically in Figure 2 [SEI 93] and each of the nine principles of team risk management is described in more detail below.

1. Shared Product Vision

Team risk management is founded upon a shared product vision which encompasses all aspects of the development program and the desire to successfully achieve that vision through the collective efforts of the team. This product vision is shared by all team members and entails a common understanding of the nature and the outcomes of the program. It is based upon the team concepts of a commonality of purpose, collective ownership, collective commitment and a cooperative interpersonal working relationship characterized by shared responsibility, mutual accountability, and individual accountability [Deming 82], [Ketzenbach 93]. While the specific nature of each individual team member's personal stake and objectives may vary, collectively all team members have a shared interest in the successful outcome of the program.

The shared product vision is formed based upon formal contractual agreements and upon a mutual understanding of the needs of the customer. Overall, the shared product vision is evidence of a common constancy of purpose in the sense of Deming [Deming 82] and a collective commitment toward mutual success in the development program.

2. Forward-Looking Search for Uncertainties

Risk management requires the management of uncertainty [Rowe88], [Charette88] and team risk management embodies an active, forward-looking search for uncertainties, and the effective control of the uncertainty and the potential for loss. Thinking toward

tomorrow, anticipating potential outcomes, identifying uncertainties, and managing program resources and activities recognizing these uncertainties are fundamental to team risk management.

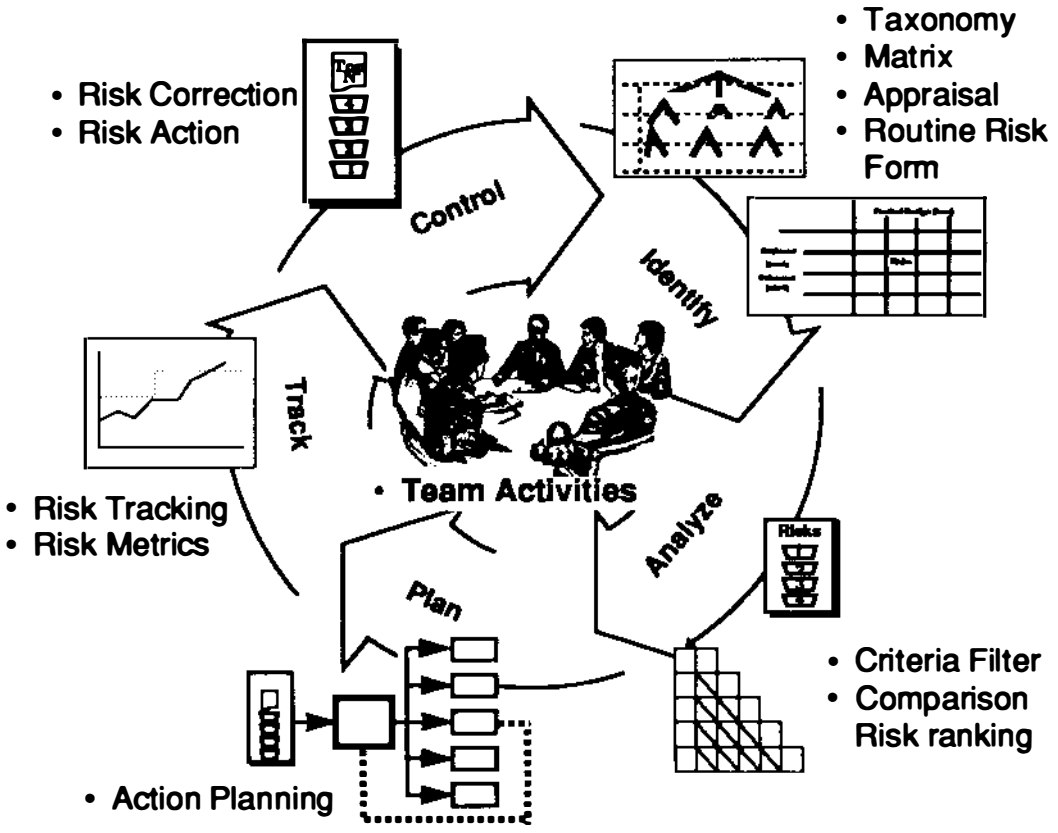


Figure 2. Team Risk Management Methods and Tools

3. Open Communications

Open, effective communication is at the heart of team risk management and involves group discussions, impromptu one-on-one and small group exchanges, and formalized information dissemination processes and tools. There are formal mechanisms to report risks to management and for management to inform program personnel. All of the forums for identification, risk resolution and management involve the free-flow of information. Communication is fostered within the key decision steps of team risk management through consensus-based decision making processes.

4. Value of Individual Perception

A key principle of the team risk management concept is that for effective team interaction, one that benefits from the synergy that results from the collective perceptions and multiple views of each individual in the team, it is vitally important to value and encourage the contribution of each individual. It is this individual voice which can bring unique knowledge, insight and perspective to the identification and management of risk.

Fundamentally, the team risk management process values individual perception and empowers individuals at all levels within the program to actively contribute to all of the steps in the risk management process. As part of the communication-centered team risk management process, it is the unique perspective of the individual that provides the knowledge and insight to recognize potential problems, risks, in the program and the expertise to support efforts in effectively dealing with these risks.

5. Systems Perspective

While the focus of team risk management is in the area of software technical risk in software-dependent development programs, the overall perspective of the approach involves a global view of the program. Software development cannot be isolated from the systems-level definition and development efforts. Within team risk management the assessments of risk and the decisions involving risks are optimized based upon a broad system context over the entire program, rather than in the isolation of the software perspective. This perspective on the software development effort includes not only the primary issues relating to customer needs, but also technical, schedule, cost, performance, and related development issues [Chittister 93].

6. Integration into Program Management

One of the crucial elements in the team risk management approach is the principle that in order to realize success in a development program, risk management must be an integral and vital part of program management. Risk management cannot be viewed as an adjunct to routine program management activities. This concept is shown graphically in Figure 3 [SEI 93] where the basic SEI risk management paradigm is embedded in the suite of program management processes and methods. Specifically, team risk management not only provides de-personalized, systematic processes for managing risk but also provides processes and associated methods that are integrated into established program management practices.

7. Proactive Strategies

The proactive character of anticipating, planning and executing program activities is a hallmark of the team risk management approach. Ultimately the management of risk culminates in decision making. Team risk management provides the foundation for informed decision making by consciously assessing the uncertainties, the potential for loss, and the opportunities afforded by anticipating rather than reacting to events. Team risk management methods embody this proactive, anticipatory approach by incorporating the awareness of the potential for loss directly into program decision making processes.

8. Systematic and Adaptable Methodology

A systematic approach that is adaptable to the program's infrastructure and culture is a key element of team risk management. The structure of the processes, methods and tools is modular, built on fundamental units of process. Each modular process unit is adaptable to an organization's unique practices, application domain, size, and program time-frame. The process units incorporate the flexibility to extend or modify details within limits that ensure the validity, accuracy and effectiveness of the approach.

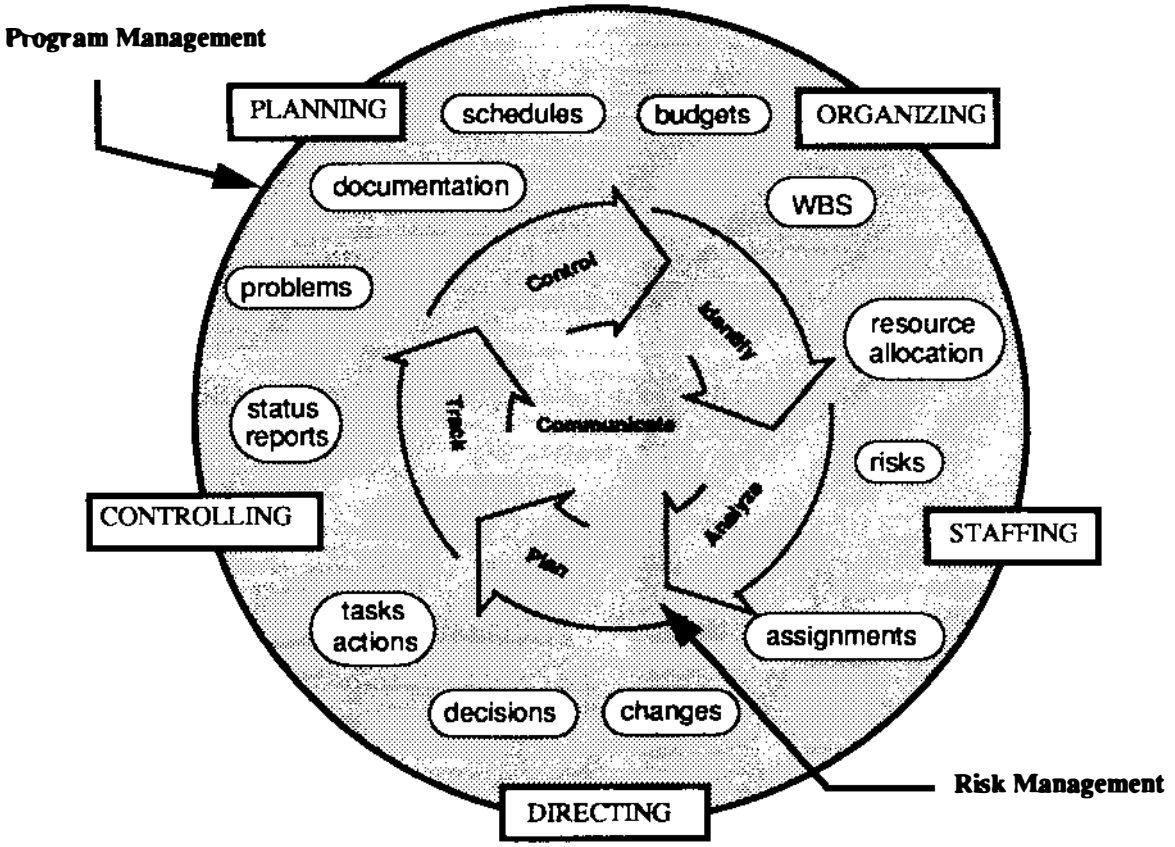


Figure 3. The Integration of Risk and Program Management

While providing comprehensive methods and tools which can coexist with established practices, the basic paradigm structure and associated methods are designed to conform to a program's existing practices, methods, and tools. For example, in implementing basic risk tracking and control, organizations can employ their routine problem report tracking and control methods through modifications to include risk information on existing report/tracking forms.

9. Routine and Continuous Processes

Fundamentally successful risk management requires a continuous vigilance, and as shown in Figure 4 [SEI 93], team risk management embraces this concept in the form of a continuous process loop that is characterized by routine risk identification and management activities that are evident throughout the life of the program. In the practice of team risk management these principles are exemplified throughout all program activities. For example, risk issues are agenda items at regularly scheduled staff meetings, discussions of risks occur routinely, and as new risks emerge or risks change plans and actions are modified and decisions are made based upon these new or changing risks.

Practices

This section discusses the application of team risk management as shown in Figure 4, to a software-dependent development program that has been awarded by the government to a prime development contractor.

The initial steps in the process involve establishing a baseline set of risks for the program. Each partner, government and contractor, conducts a baseline risk assessment to define the risks that are associated with their respective organizations. Currently, the baseline risk assessment is conducted by the SEI to initiate the overall team risk management process, but as the methods and associated tools are finalized and released, the assessment will become part of the organization's capability to institutionalize the practice of team risk management on the program.

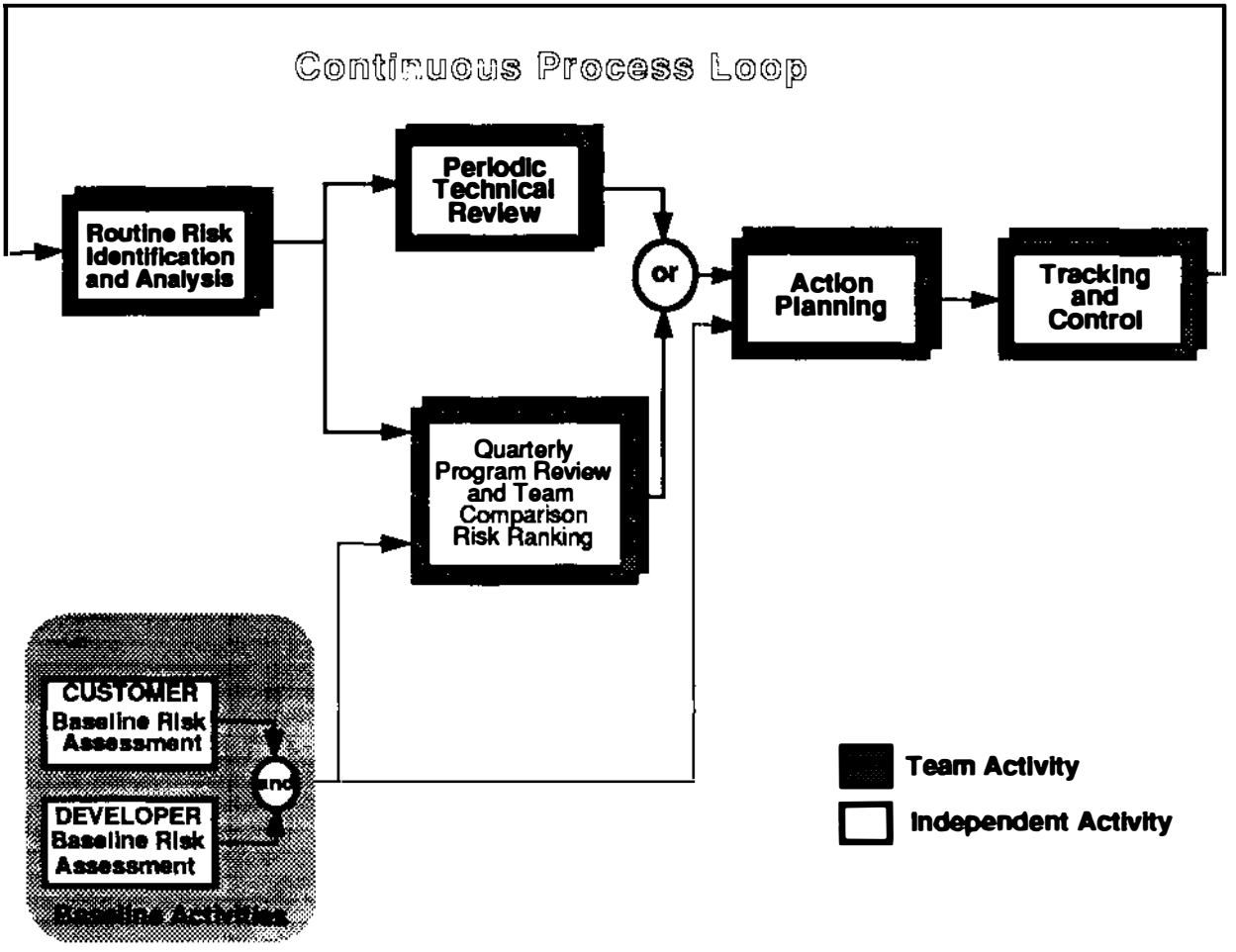


Figure 4. The Team Risk Management Process

1. Baseline Risk Assessment Activities

The key process steps involved in a baseline risk assessment are shown in Figure 5 [SEI 93]. All of the baseline risk assessment activities are conducted by a trained risk assessment team.

The initial step in a baseline risk assessment, the ***participants' orientation***, provides an overview of the process to all of the participants involved and prepares them for their roles in that process. This presentation is the start of the "buy-in" that is so important in introducing change into an organization.

Following the participants' orientation, multiple ***group sessions*** are conducted. Each group session consists of an interview and a risk evaluation activity. A taxonomy based questionnaire and a non-judgemental, non-attribution group interview technique are employed for the interview. The taxonomy-based risk identification process is the culmination of an extensive research, development, and testing effort and provides an efficient structure for identifying risks in a software-dependent development program [Carr 93].

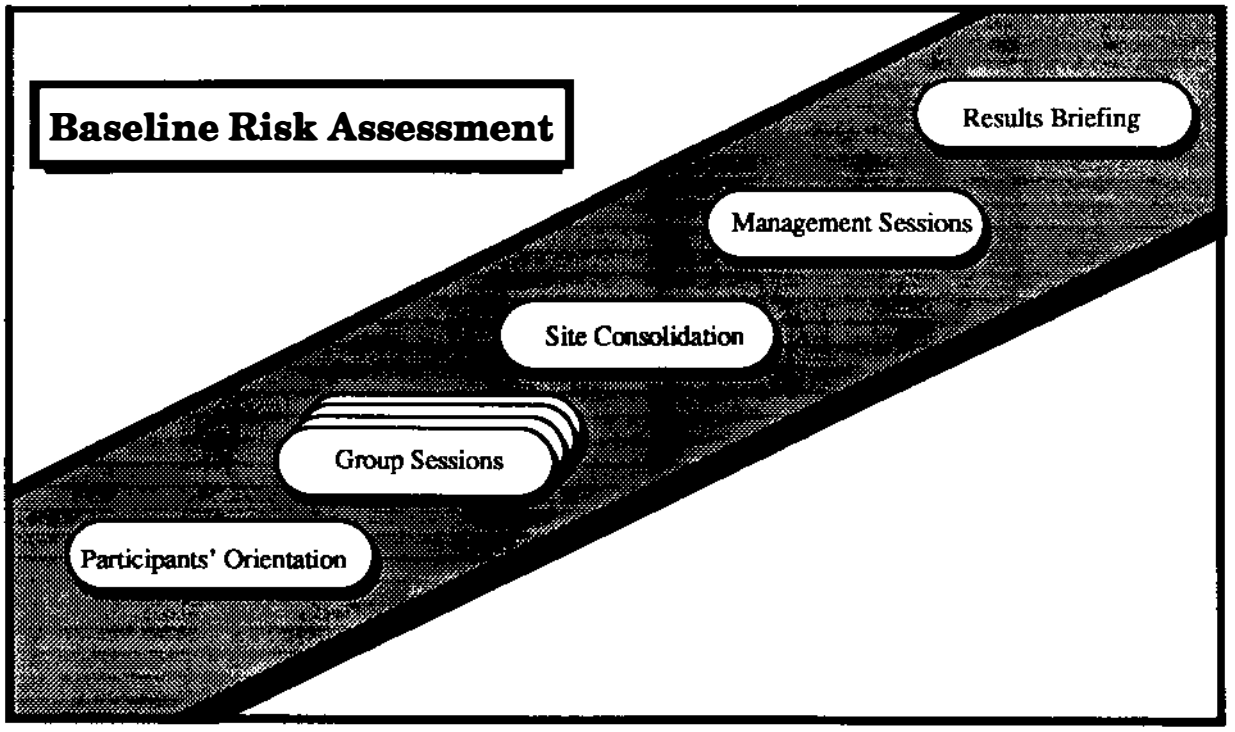


Figure 5. Baseline Risk Assessment Process Steps

The interview process which is facilitated by the risk assessment team, embodies a number of important team characteristics:

- **group interviews:** The exchanges that occur within the interview provide the foundation for additional communication outside of the group interview. In large programs, this forum is often the first conversation between many of the

participants, despite the fact that they may have been working on the same program for a number of months.

- **peer groups:** Peer grouping facilitates communication between participants. As a result, there is a common language, common perspective, and mutual understanding within the group and each participant can identify with similar "problems" and issues.
- **non-judgemental responses:** Assessment team members do not directly or indirectly make judgements on the discussions that occur or risks that are identified. Rather there is a shared sense of "getting to the truth."
- **non-attribution:** Risks and related information emerging from the interview are not attributed to the group or any individual in the group. The conversation surfaces important, often suppressed information.
- **free flow of information:** While structured around the taxonomy based questionnaire, the interview involves a free flow of information where responses to the questions are sought, not "correct answers."

Thus, the professional team atmosphere, the structure and the free information flow of the interview, complemented by the non-attribution, non-judgemental characteristic of the process, overcomes the reticence and anxiety often evident in audit or evaluation interviews. This process empowers each group to challenge their collective understanding of the risks associated with their program. The knowledge and expertise contained in the taxonomy questionnaire combined with the de-personalized interview process enables participants to explore these uncertainties as professionals.

In contrast to an external audit, where there are judgements and recommendations rendered by the audit team, an atmosphere of mutual cooperation is evident in an SEI baseline risk assessment. There is a sense of constructive support provided through the assessment team members' guidance and a strong sense of working together toward a common goal.

The evaluation activity within the group session involves the individual assessment of the significance, likelihood and time-frame associated with each risk identified in the group session. In addition, a selection of the five most important risks is made by each participant. These evaluations are the initial analysis steps of the risk management process, and the resulting information is used to facilitate management decisions on risks to the program.

The ***site consolidation*** activity merges the multiple group session data into a consistent package to prepare for the establishment of program priorities for the identified risks. Within the site visit consolidation step, significant decisions required to compile risks are based on consensus among the assessment team members.

The ***management sessions*** consist of two activities, management review and management comparison risk ranking. Collectively, these activities are the mechanism to quickly focus on the most important risks and place them in a management-defined priority order.

The management sessions are designed using risk management principles and, in particular, incorporate steps that facilitate and enable a shared product vision, open communication, the expression of individual perception, and a systems perspective. The management review process includes an implementation of the Nominal Group Technique [Schulles 88] to enable management to identify the most important risks to the program and to begin to establish a common management understanding of each of the most important program risks. The management comparison risk ranking activity is fundamentally a consensus-based process which arranges into priority order the most important risks, as identified in the management review session.

In the management comparison risk ranking activity, the ranking of the risks is accomplished using the comparison risk ranking method [FitzGerald 90] and a consensus-based decision making process. In this activity each risk on the list is compared, pairwise, to every other risk relative to the criterion of "importance to the program," i.e. which has the greater risk exposure, which should be allocated resources, which should be given management attention, etc. Through discussion on each of the pairs, the risk in the pair that is more important to the program is decided by a consensus among the managers.This process continues through the entire list and the risks are prioritized based upon the total of all comparisons. Aggregate effects of risks, extreme event risks (low probability but high impact), and all identified risks are addressed as part of the continuous management (planning) processes. Risks are added to the list, as required, throughout the life of the program.

While consensus is sought within the management sessions, the consensus-based decision making processes are defined such that the program manager maintains the authority and the responsibility for the outcome. Ultimately, the results of the process belong to the Program Manager. The consensus-based processes are effective methods to foster a shared vision, a systems perspective, open communications, and the formulation of proactive strategies among management personnel.

The final step in the baseline risk assessment is a presentation of the results, without attribution to any group or individual. This ***results briefing*** is conducted as a formal presentation to all program personnel who participated in the assessment process. While this step is the conclusion of the baseline risk assessment, this presentation is also the forum to initiate the continuous processes of team risk management.

2. Continuous Process Steps

As shown in Figure 4, the results of the baseline risk assessments are used in the team comparison risk ranking, which is conducted in conjunction with quarterly program reviews. The process of the team comparison risk ranking session results in the prioritizing, or re-prioritizing, of the program-wide "vital few" risks to the program. This program-wide vital few list, herein referred to as the program-wide Top N list, is representative of the Pareto-like approach of managing the "vital few" [Juran 89] at the highest management levels of the program.

The team comparison risk ranking process, which is initially conducted after the baseline activities and approximately quarterly thereafter, is representative of the important role that a working team environment, founded in effective communication, plays in the implementation of team risk management. This forum is an especially effective mechanism for exchanging the individual perspectives and priorities that each partner, either government or contractor, brings to the process of developing a software-dependent product.

Prior to the initial team comparison risk ranking session, the most important risks identified in each of the partner's baseline risk assessments are brought together to form the program-wide most important list, the "Top N" list. This program-wide Top N list is a shared list between the government program office and the contractor. The selection of the risks that are shared is based upon the criteria established by the respective program managers, government and contractor.

The participants in the team comparison risk ranking session are the government program manager and selected personnel from the government program office and the program manager and selected senior staff of the contractor. The same method, the consensus-based comparison risk ranking method, that was used in the baseline process is used to accomplish the ranking.

The outcome of this process is a prioritized program-wide Top N list of risks. The team comparison risk ranking process, based upon discussion and consensus, and the program-wide Top N list are the formal inter-organization communication methods between government and contractor regarding risks in the program. Team comparison risk ranking sessions are held approximately quarterly and at major milestones throughout the life of the program. As an integral part of continuous team risk management, these formal exchanges are the primary facilitators of the routine inter-organizational dialogue on risks.

The remaining continuous process steps involved in team risk management are conducted within each partner organization and include the continuous risk identification and analysis processes and the planning, tracking and control steps of the risk management paradigm. In addition, risks and risk related program activities are reviewed periodically during technical reviews, management interchanges, and formally at quarterly program reviews.

The routine identification of risks involves personnel at all levels of the organization. This identification process as well as all of the other steps in the continuous process loop shown in Figure 4 are representative of a risk aware culture. In continuous team risk management, risk issues are an integral part of the routine development activities of the program. Risks and the issues related to risk management are an agenda item at staff meetings throughout the life of the program. As new risks emerge, or known risks evolve, plans and actions are modified and decisions are made based upon this information. The tracking and control processes provide methods that enable anticipatory actions to deal with risks. Fundamentally, continuous team risk management becomes an integral element in all of the program's development activities.

Team Risk Management is a self-sustaining methodology that does not depend upon outside organizations for continued successful operation and process improvement. Consequently, through the adoption of team risk management as an integral part of routine program management processes, organizations can achieve self-sufficiency in software risk management.

Implementation Challenges and Observations

There are many challenges in applying the team risk management process within an organization, but the central issue is one of establishing a team-oriented environment characterized by a risk ethic [Kirkpatrick 92]. Toward this end, the Risk Program within the SEI has been maturing its risk management methods by applying the methods to government and industry software-intensive development programs.

While most managers feel that they are managing risks [Kirkpatrick 92] and generally successful managers are good risk managers [Boehm 91], the preliminary results of the Software Engineering Institute's risk management program demonstrate that few organizations have explicit policies for risk management and most organizations that have addressed risk issues have done so through undocumented, ad hoc or de facto policies [Kirkpatrick 92].

It is especially evident that software risks are among the least measured or managed risks in a system and that in technology areas more familiar to most DoD program managers (e.g. aerodynamics, propulsion, air frames) the risks are well managed. Generally, managers are effective in managing the risks associated with the technologies they know and consequently overall risk management activity is very dependent on individual judgement and experience [Kirkpatrick 92].

During the field testing and implementation of team risk management, the introduction of a structured process for identifying, analyzing and generally managing software technical risks has altered the perception of risks within an organization and expanded the awareness of these issues. Even in the programs where "problem" management was used as a risk management vehicle, the incorporation of the structured, depersonalized team risk management process coupled with pro-active planning strategies, has resulted in enhanced risk management capabilities.

In the continuing application of the risk management process to large software development programs, the most dramatic effect has been in opening the communication channels for dialogues within organizations relating to risks and risk management. The non-attribution which exists throughout all steps of the process has proven extremely effective in fostering openness in risk discussion. This effect has been observed in peer group interactions as well as in joint management sessions.

The impact of the application of team risk management and the basic risk management process to government and corporate organizations is evidenced in the evaluations and comments provided by organizations participating in the testing of the methods. A sample of the comments include:

"brought out many risks that had not been previously identified"
"more than expected risk identifications"
"opportunity to consider some areas... not focused on before."
"comfortable setting to express concerns"
"comprehensive"

Through team risk management practices, program personnel are empowered with methods and tools that capitalize on the characteristic that, collectively, teams (working together based upon the nine principles of team risk management) actually possess more knowledge, think in a greater variety of ways, and are more effective than the totality of the team members working as individuals. Institutionalizing the team risk management approach and enabling organizations to realize self-sufficiency in risk management is the major challenge and objective of the Software Engineering Institute's team risk management project.

Summary

Team Risk Management provides the framework for rational decision making through forward-looking, team-oriented and pro-active processes which identify and manage risks. These practices enable software development organizations to deal effectively with exposure to hazards that can lead to failures in meeting customer needs, to product deficiencies, and to delayed product delivery. Team risk management is a vital ingredient toward ensuring overall customer satisfaction and total product quality.

In today's fast-paced environment which is increasingly characterized by rapid change, a very short time-to-market, and often very short duration market window, it is imperative to assess both opportunities and what can go wrong. To be informed in decision making and to address the uncertainty in a technically challenging and economically competitive environment it is necessary to effectively manage risk. The broad perspective and related synergy provided by team risk management is an effective foundation for realizing success in today's business environment by building the "right" product, building the product "right," and delivering the product at the "right" time.

Acknowledgments

A number of individuals from the team risk management project either directly or indirectly contributed to this article. We extend our appreciation to Audrey Dorofee, Julie Walker, and Ray Williams for their comments on this article and to all of the members of the Software Engineering Institute's risk management program whose efforts influenced the content and direction of this work. We would also like to thank the reviewers of this article for their helpful suggestions. Most importantly, we extend special appreciation to the personnel throughout all of the risk program's client organizations whose support and participation contributed significantly to the successful development of the team risk management process.

References

[Akao 90] Akao, Yoji. *Quality Function Deployment: Integrating Customer Requirements into Product Design.* Cambridge, Mass.: Productivity Press, 1990.

[Boehm 91] Boehm, Barry W. "Software Risk Management: Principles and Practices." *IEEE Software*, 8, 1 (January 1991):32-41.

[Carr 93] Carr, Marvin; Konda, Suresh; Monarch, Ira; Ulrich, Carol; & Walker Clay. *Taxonomy Based Risk Identification.* (CMU/SEI-93-TR-6). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1993.

[Charette88] Charette, Robert N. *Software Engineering Risk Analysis and Management.* New York: McGraw-Hill, 1988.

[Charette 90] Charette, Robert N. *Application Strategies for Risk Analysis.* New York: Multiscience Press, 1990.

[Chittister 93] Chittister, Clyde; & Haimes, Yacov Y. "Risk Associated with Software Development: A Holistic Framework for Assessment and Management." to be published in the *IEEE-SMC Transactions.*

[Deming 82] Deming, W, Edwards. *Out of the Crisis.* Cambridge, MA: Massachusetts Institute of Technology, Center for Advanced Engineering Study, 1982.

[Dorofee 93] Dorofee, Audrey. "Risk Process Model" to be presented at the Software Engineering Symposium, Pittsburgh, Pa., August 23-26, 1993.

[FitzGerald 90] FitzGerald, Jerry; & FitzGerald, Ardra F."A Methodology for Conducting a Risk Assessment. "*Designing Controls into Computerized Systems.*" Second Edition, Redwood City, CA: Jerry FitzGerald & Associates, 1990.

[Juran 89] Juran, J.M. *Juran on Leadership for Quality.* New York, NY: The Free Press, A Division of Macmillan, Inc., 1989.

[Juran 88] Juran, J.M. *Juran 's Quality Control Handbook: Fourth Edition.* New York, NY: McGraw-Hill Book Company, 1952,1979, 1988.

[Katzenbach 93] Katzenbach, Jon R.; & Smith, Douglas K. "The Discipline of Teams." *Harvard Business Review*, XXX, (March-April, 1993):111-120.

Kezsbom 89] Kezsbom, Deborah S.; Schilling, Donald L.; & Edward, Katherine A. *Dynamic Project Management.* New York, N.Y.: John Wiley & Sons, 1989.

[Kirkpatrick 92] Kirkpatrick, Robert J.; Walker, Julie A.; & Firth, Robert. *Software Development Risk Management: An SEI Appraisal* (SEI Technical Review '92). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

[Schulles88] Schulles, Peter R. The Team Handbook: How to Use Teams to Improve Quality. Joiner Associates, Inc., 1988.

[SEI 92] Software Engineering Institute. "The SEI Approach to Managing Software Technical Risks."*Bridge*, (October 1992):19-21.

[SEI 93] Internal technical documentation - publication of an SEI Technical Report on Team Risk Management is pending.

[Vesey 92] Vesey, Joseph T. "Time-to-Market: Put Speed in Product Development." *Industrial Marketing Management*, 21, (1992): 151-158.

[Winner 88] Winner, Robert I.; Pennell, James P.; Bertrand, Harold E.; & Slusarczuk, Marko M. G. *The Role of Concurrent Engineering in Weapons Systems Acquisition* (IDA Report R-338). Institute for Defense Analysis, December 1988.

Implementing Software Process Improvement

Karen S. King
Credence Systems Corporation
9000 SW Nimbus Ave.
Beaverton, OR 97005
Tel (503) 520-6452
FAX (503) 520-6400

Karen King is the project leader of Software Quality Assurance at Credence Systems Corporation in Beaverton, Oregon. She received her BS in Electrical Engineering from Rice University, and has over ten years experience in software and hardware evaluation within the test and measurement industry. She is chairman of the Software Engineering Process Group (SEPG) at Credence.

Keywords

Software Process Improvement, Deming, Software Engineering Proces Group, software quality, SEI capability maturity model, Crosby, measurement, metrics

Abstract

The SEI capability maturity model describes a model for increasing software productivity, as well as increasing software quality, by improving software processes. This paper describes the formal procedure of how to implement Process Improvement, as defined by Deming. It relates the experiences of the first year of a Software Process Improvement program at Credence Systems Corporation. Within that year, the organization has reversed the trend of a rising backlog of bugs, while maintaining a constant investment in fixing bugs.

The process of process improvement begins with realizing a need for change, and never ends. This presentation describes the process of how we convinced management that there must be resources dedicated to software process improvement, what changes were requested, how the changes were implemented, and the results of the changes. In addition, I discuss how the first process improvement cycle pointed out the need for the SEPG to work with all software engineers to define and implement processes for software product development, and how that work is continuing to the present.

The third major subject discussed, is common reasons why improvement programs fail. Our improvement program ran into many of these problems, such as trying to change software design processes by "management decree", trying to make changes without a commitment of additional resources from upper management, and trying to make changes within "fire-fighting mode". This paper clearly outlines each of the problems, as well as others, and how we solved them. The goal is to educate people in the audience, and to help them avoid these common pitfalls.

Implementing Software Process Improvement

Introduction

Throughout the years, almost all organizations have struggled with the problem of how to develop new products faster, with fewer defects, and with constant resources. Many times, management has tried to solve the problem by reading about a new method, then dictating to the masses that starting today, things will be done differently. Almost universally, what results is that people will make the changes in the beginning to the minimum standards, and then efforts will decrease afterwards. In many organizations, what results is a "management decree" which is issued at regular intervals, with little process change visible in the interim. How can this cycle be broken? What does it take to effect actual changes within an organization? This paper will attempt to answer these questions by describing the Crosby Problem Solving Method, and by describing the experiences at Credence Systems Corporation in Software Process Improvement.

Process Improvement Model

One common model for process improvement, is the Crosby Problem Solving Method, which is described in much of Philip Crosby's work. This method is used to successively solve problems in order to achieve Continuous Process Improvement. The Problem Solving Method is described by the following five steps:

1. Define the Situation
    a. Set Goals to specify the situation which needs to be changed
    b. Collect data on what is being done now relative to the goals
2. Fix
    a. Implement a temporary patch to keep the organization going
3. Identify the Root Cause
    a. Analyze data to determine areas which could be improved to meet the goals
4. Take Corrective Action
    a. Propose process changes which could be used to meet the goals
    b. Plan integration of changes into process
    c. Implement improvements by changing the process
    d. Use Improvements by operating under the new process
5. Evaluate and Follow Up
    a. Measure change against the goal

The steps of the Problem Solving Method will be illustrated in the following Process Improvement example.

DEFINE THE SITUATION

Set Goals

One of the biggest concerns within the Credence Software Organization was that there were too many defects introduced within the software development process. The number of defects led to an increasing backlog of unfixed bugs, even though heavy resources were being spent fixing defects. The goal was set to reduce the backlog by 25% without a commitment of increased resources for fixing defects. The timeframe for achieving the goal was by the end of the release twelve months from then.

Collect Data

The area to be measured is that which pertains to the goal, in this case, data about defects. The Software Engineering Process Group (SEPG) collected data on the history of the backlog (See Figure 1). In addition, data was collected on the rate which defects were submitted, the rate which defects were fixed, and the rate of accrual of software fix costs. The raw data was used to persuade the organization that the only way to meet our goal was to prevent defects from being introduced.

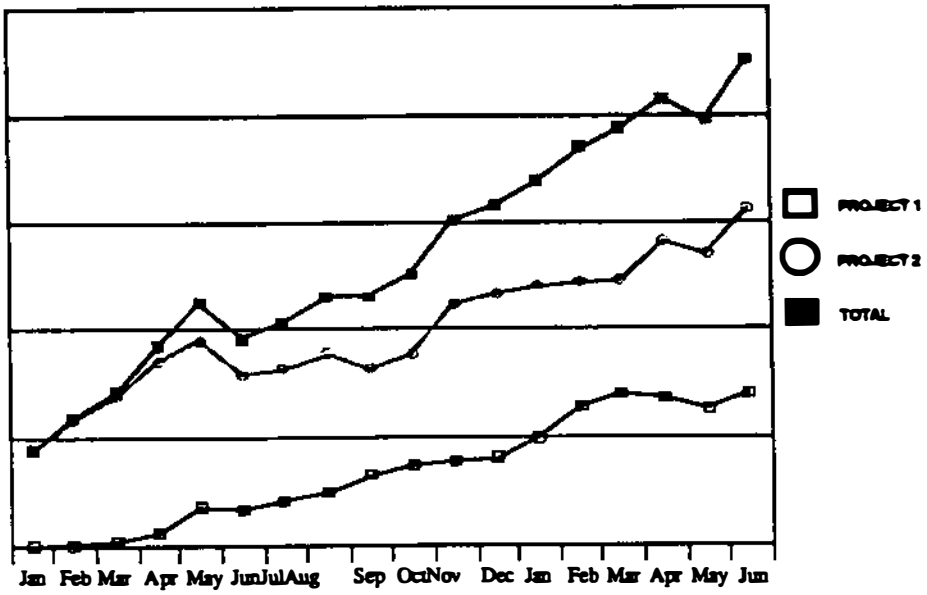


Figure 1: Chart of Software Defect Backlog

FIX

Implement a Temporary Patch

Since the main problem is increasing backlog, the temporary solution to the problem was to devote more resources to repairing defects. Since this solution was to devote more resources to decreasing the backlog, it was obviously only a temporary measure. The long-term solution to the problem was to begin preventing defects from being produced.

IDENTIFY THE ROOT CAUSE

Analyze Results

The SEPG worked to determine the major cause of software defects, by both number of defects, and time spent correcting the defect. Because much of the literature describes the difference in cost of defects introduced in different phases, we decided to divide the defects into Requirements defects and Implementation defects. All defects were sorted on the basis of whether the defect was caused because the coder didn't know what to do (Requirements defect), or whether the coder made a mistake. In addition, each of those two categories were divided into defects which were found internally and those which were found externally. With these four categories, Requirements-Internal (RI), Requirements-External (RX), Implementation-Internal (II), and Implementation-External (IX), a Pareto chart was drawn to show the percentages of defects attributed to each type (See Figure 2). There seemed to be little difference between the number of defects attributed to Requirements problems, versus the number of defects attributed to Implementation problems. The next step was to measure the amount of resources spent correcting each defect type (See Figure 3). This chart was generated by taking data on the average amount of time spent to repair a defect of the given type, then multiplying that number by the number of defects repaired of the given type. It was found that 71% of all time spent repairing defects could be attributed to 45% of the defects, which were caused by requirements problems.

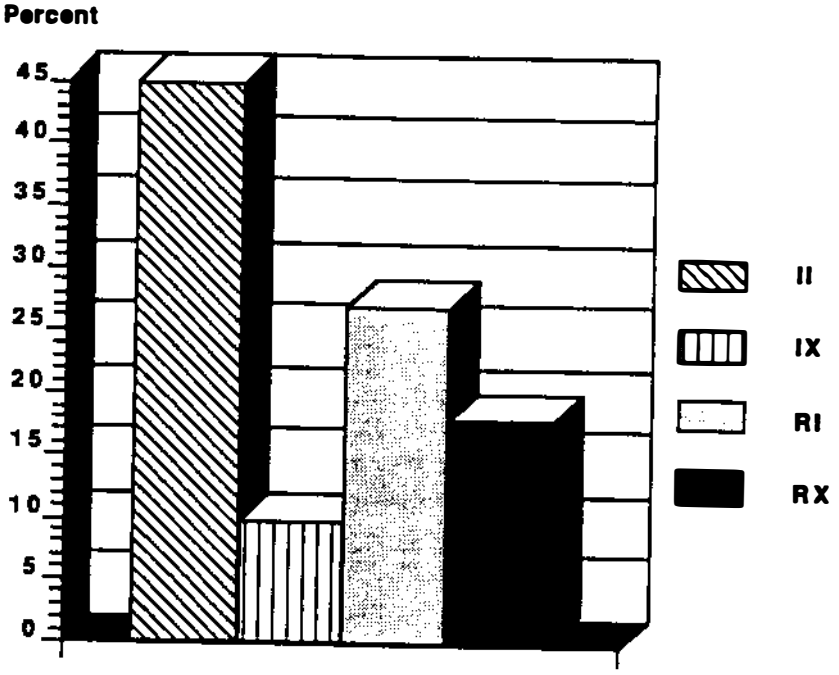


Figure 2: Pareto chart sorted by number of defects found

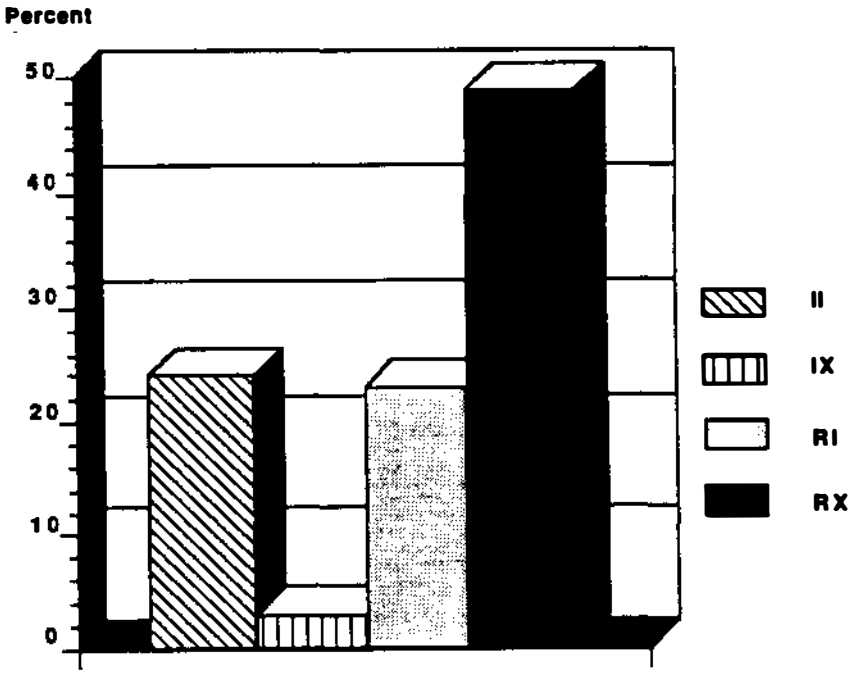


Figure 3: Pareto chart sorted by cost of defects found

TAKE CORRECTIVE ACTION

Propose Process Changes

Since the analysis showed that the most error-prone stage of software development within our organization was the requirements phase, the SEPG brainstormed on what kinds of problems caused requirements defects. There were three areas identified which contribute to requirements defects, which are as follows:

1) Engineering knew the requirement, but the coder didn't

2) Marketing or Applications knew the requirement, but the coder didn't

3) Customer knew the requirement, but the coder didn't

The SEPG worked on each cause to determine what process changes could be made to correct each defect type.

In the case where Engineering knew the requirement, a possible solution would be to create system architecture documents which detail the system requirements for each separate system component. In addition, whenever a new feature is proposed, there should be information detailing any particular internal requirements which may be imposed on the feature.

In the case where Marketing or Applications knew the requirement, a possible solution would be to have an Applications Engineer write down their understanding of the Feature Description and Feature Requirements. That description would be reviewed with the other Applications Engineers along with Software Engineers to ensure the requirements list is clear and complete.

In the case where the Customer knew the requirement, a possible solution would be to have the Customer review the Feature Description and Feature Requirements written by the Applications Engineer.

Plan Integration of Changes into Process

All the proposed changes are new processes within the Requirements phase. Since there are several activities which involve documenting Requirements, the SEPG decided to create a new document known as the POR Feature Worksheet (See Figure 4). The POR Feature Worksheet documents the history of each new feature, along with a description, and a list of both internal and external requirements. In addition, the POR Feature Worksheet contains early estimates on the time required to develop and test the feature. The process for developing the Worksheet starts with an Applications Engineer documenting the history, description, and requirements. It is then assigned to a Software Engineer, who determines the internal requirements, and estimates of the amount of time required for each phase..

Implement Improvements

Once the process change has been defined, it must be implemented. This step involves deciding what changes in environment are necessary to allow the changes to be successful. It also includes educating everyone involved of what the process change is, and why it's being changed. In many cases, people must be educated in the value of the change, and how it fits into the goal.

In this example, the SEPG listed reasons why previous process improvement plans were not effective. The list generated is as follows:

1) There was no plan to educate engineers on the process changes

2) There was no extra time allowed to implement the new techniques

3) There were constant pressures to meet customer commitments

4) The organization tried to change the entire process at once

Once we were aware of the problems, we worked to figure out solutions. The solutions suggested were as follows:

1) a) Have software meetings which provide training on specific software processes, as well as general training on Software Improvement Methodologies. These meetings also provided an opportunity to share new measurement data, which marked progress the organization had made towards meeting the goals.

   b) Create templates for all required deliverables. Templates simplified process steps by providing a fill-in-the-blank method for creating deliverables.

   c) Create examples for all required deliverables. Examples help verify that the template is easy to use, and provides verification that the template is correct. The example also provides clarification of what information should go in which blank.

2) Add 10% additional time to the expected schedule. This extra time is provided to allow people time to learn how to properly use the new process steps.

3) Require Sales to consult Management before making sales commitments. When Sales does not consult Management, there is often the situation where Engineering must meet a deadline which was not based on actual time required. In those cases, Engineering must take shortcuts, and often compromise the development process, as well as the quality of the final product.

4) Only change items indicated in the Goals statement.

POR Feature Worksheet Number:

Feature Name:

$Author$ $Revision$ $Date$

Background

Feature Description

Customer Feature Requirements

Internal Feature Requirements

Estimate of Engineering Development Time (in Man Days)

Design time estimated by:

Requirements		Specification:	
	POR Feature Worksheet		SFRS Document
	SFRS Document		Headers of Code
	POR Document		Manual documentation
Design:		Implementation (Coding)	
	SDDS Document		New code
	SFRS Document		Old Code, new portion conforming to coding standards
	Code headers		Old Code, entire file brought up to coding standards
Reviews: (Select all that are required)		Release Notes	
	POR Feature Worksheet Review		To Be Written by:
	Requirements Review	Application Note	
	Specification Review		Revision Required to App. Note #
	Design Review		New App. Note to be written by:
	Code Review		
	Postmortem Review		
Example Programs			
	Included in Application Note		
	Update existing examples programs		
	To Be Written by:		

Estimate of Quality Assurance Time (in Man Days)

Evaluation time estimated by:

	Use existing acceptance test suite.
	Add tests to current acceptance test suite.
	Use existing unit test suite.
	Develop new unit test suite.
	Develop new "clear box" test suite.

Estimate the Manuals Development Time (in Man Days)

Manuals development time , estimated by:

Support Requirements

Figure 4: POR Feature Worksheet

These recommendations could be grouped into two categories, those which could be solved within the Software Engineering Organization, and those which required Management intervention. The SEPG created a proposal to management which outlined each of the previous steps. In addition, the proposal outlined the requests to management (items 2 and 3), the internal changes, as well as the goals which will be achieved if the requests were granted. In this way, Management was educated about what process changes were to be made, as well as the reasoning behind each request.

The next step was to educate Engineering. The same presentation was made, which explained why the changes were being made. All requests were accepted, and were implemented by the SEPG, along with Software Management. Everyone agreed that starting that day, no features would be implemented without following the Feature Selection Process.

Use Improvements

Once the improvements have been implemented, the organization must spend at least six months using the process as newly defined. During this time, there should be intermediate measurements to verify how well the process is being accepted, as well as measurements of how the organization is progressing towards the goal.

In this case, the new Feature Selection Process was implemented, although there were some problems. There had been no training of the Field Applications Engineers, so the reviews of the POR Feature Worksheet were not completely successful. In addition, not all people involved in the release were invited to the meetings. In general, there will almost always be some implementation problems, which will end up being resolved within the next development cycle. Evaluating the process changes, and improving them in the next iteration is considered to be an important part of the Problem Solving Process.

EVALUATE AND FOLLOW UP

Measure Results

Once the Improvement Cycle is complete, the SEPG should measure the final result, to determine the effect of the change. At that time, it will be determined whether or not the goal was achieved. In addition, all process measurements should be recorded at this time to help decide the next area of improvement.

Repeat Process

The Problem Solving Process should be repeated by examining a new situation, based on the new process measurements. Repeating the cycle ensures that the organization is implementing a Continuous Process Improvement program.

COMMON PROBLEMS AND SUGGESTIONS

In working through a complete Process Improvement Cycle, we discovered several recommendations, or helpful hints, which may be useful to other organizations considering a process change.

1. Management Commitment

There are two areas in which Management needs to demonstrate its commitment to process improvement. First, Management must be able to commit internal resources towards process improvement. Without resources, it will be extremely difficult to successfully make significant process changes. Second, Management sometimes must exert influence on external organizations to change processes which may affect the Software Development Process. For example, during the requirements review process, it's important to involve Internal and External Customers in reviewing Requirements Documents. Usually, the easiest way to implement that change is if the need is explained to Management and to the external organization.

2. Devoted Group to Oversee Process Improvement

In order to make significant improvements, there must be a group set up to oversee Process Improvement, in this case, the SEPG. The SEPG is a group of software engineers who know the problems that must be solved, because they are doing software design on a daily basis. The group works to survey public opinion, and works on solving the most serious problems, as described by the software engineering community. Since the SEPG members themselves deal with the problems on a daily basis, it seems appropriate that they are the proper organization to devise solutions to the problems.

3. Standards

Without standards and process descriptions, the process change is open to differences in interpretation. Additionally, there is no standardization of how the process is to be implemented, and; in general, nothing is changed when the process is not well-defined. In order to be successful, there should be a written description of how the process is to be performed, as well as a standard of how the deliverable should look.

4. Measurements

Without measurement, there is no visibility of progress towards the goal. It helps organizations to be able to see that there is progress, because often, change is not easily discernable during intermediate stages. Another need for measurement is to have an objective way of measuring that the goal has been achieved. Without concrete measurements, it may be difficult to determine whether the goal was made.

5. Too many large changes are tried at once

When many large changes are made at once, the organization usually cannot handle that volume of change. When people are not focused on just a few key changes, then their effort becomes fractured, and nothing is accomplished. In general, it is better to make too few changes at once, instead of making too many, because with a few changes, there is at least some improvement. If there are many changes to be made, they should be segmented into multiple phases.

6. Processes are dictated to an organization

Processes should be designed by the organization itself. The people who perform the work on

a daily basis are the best to create process definitions which are likely to function within the organization. The processes should be reviewed by management and the SEPG, but mainly to verify that the steps fit together correctly.

7. Expectations of immediate change

Process improvement does not happen overnight. In general, it takes 1 1/2 to 2 years to implement a major change. This equates to at least three design cycles. The first cycle is used to try out the change. The second cycle is used to practice the change. The third cycle is used to develop consistency. If changes occur at a faster rate, the organization will revert to the old methods when there is a crisis, which is the time when the change is most necessary.

SUMMARY

Throughout this process, we have found the Problem Solving Process to be a very important tool in Process Improvement. In addition to the major problem described here, we have also used the same process in a less formal method to solve smaller problems. Even within the scope of smaller problems, the main steps are important checkpoints to ensure that the solution found is complete, and not just an elaborate patch. Using this process, we believe that we have achieved our first short-term goal, which was to stabilize the organization so that we could begin a phase of process definition. This phase is currently underway, and we are generating process flow charts to describe how software is designed at Credence. Our next short-term goal is to achieve SEI capability maturity level 2, and then to continue through the different levels. Throughout these improvements, our long-term goal is to develop processes which support the design of zero-defect software.

REFERENCES

1. P. Crosby. "Quality is Free: the Art of Making Quality Certain", New American Library, 1979
2. P. Fowler and S. Rifkin. "Software Engineering Process Group Guide", Research Access, 1990
3. W. Humphrey. "Managing the Software Process", Addison-Wesley, 1989
4. M. Paulk, B. Curtis, and M. Chrissis. "Capability Maturity Model for Software" Software Engineering Institute, 1991
5. Pietrasanta, Al. "A Strategy for Software Process Improvement", Pacific Northwest Quality Conference Workshop, 1991

Report on the Quality Wave Model

Becky Joos
Software Engineer
Motorola
6501 William Cannon Drive West
Austin, Tx 78735-8598
M/D OE112
512-891-3617
beckyj@pets.sps.mot.com

Ilona Rossman
Software Engineer
Motorola
6501 William Cannon Drive West
Austin, Tx 78735-8598
M/D OE112
512-891-3194
ilonar@oakhill.sps.mot.com

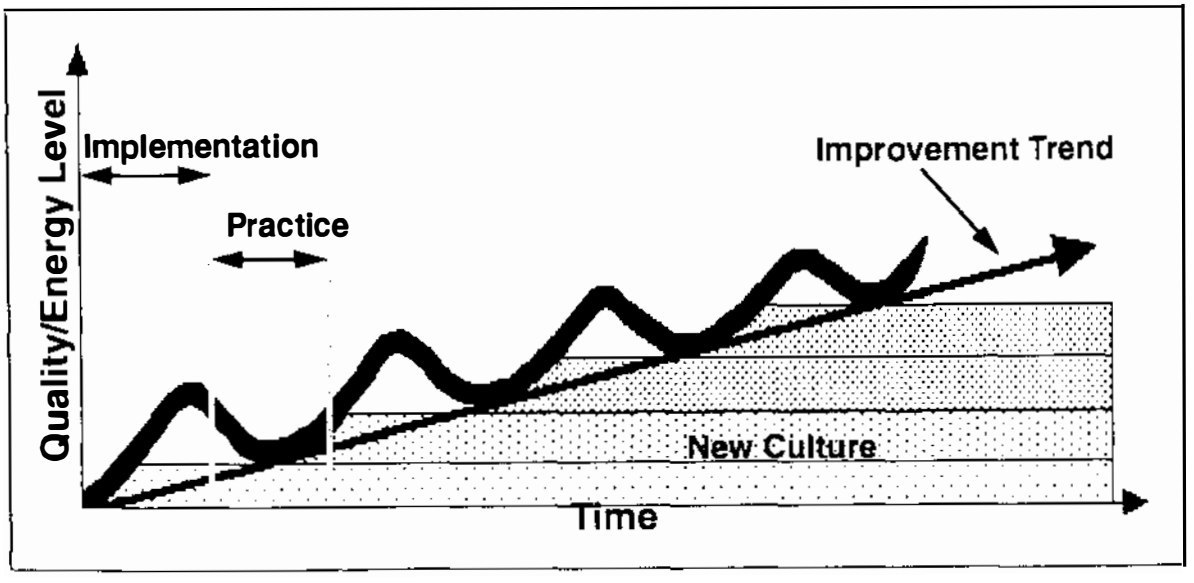
Abstract: This paper describes the waves of change a software organization goes through when implementing a quality program. We have found that our quality improvement process can be mapped via a wave-like graph which we call the Quality Wave Model (see Figure 1). The crests and troughs of the waves correspond to the improved level of quality and the acceptance and integration of the program in the organization. The wave model shows that implementing a quality program is not something that can be enacted all at once but must be rolled out over a period of time depending on the particular software organizations level of readiness.

Each wave represents a step in improving the software quality program. This paper discusses the wave in detail, what initiates it, what makes it crest, and what makes it trough. Insight is provided for the quality staff and management to prepare for the ups and downs so that they will be able to stop the descent of the waves to insure continuous improvement as well as allow enough lull between waves for new ideas to become a part of the software culture.

Keywords: quality, model, change

Introduction: The authors work in the RISC Software Department (RISC Division, MMTG, Austin, TX). The department is responsible for developing state-of-the-art RISC compilers, tools, and simulators. They have recently announced four new software support packages for the PowerPC(tm) Family of RISC microprocessors - the Software Development Package (SDP); the C and FORTRAN compilation systems; and the Architectural Simulator. These tools enable developers to reduce the time needed to reach the rapidly expanding PowerPC market, ensuring a competitive edge in application performance.The department is less than 5 years old and has made great strides in creating a quality software development organization.- Ilona Rossman is manager of the Test & Quality Assurance Group. Becky Joos is the department quality

Figure 1. Quality Wave Model



champion and has a Ph.D. in computer science.

The Quality Wave Model: This model proposes that the activity in building a quality program follows a wave pattern with crests of high energy levels, activity, and enthusiasm and troughs of adjustment, review, and acceptance (Figure 1). After each upward drive of enthusiastic activity there is a lull period when new concepts are put into actual practice and become part of the day to day routine. In a continually improving program the upward motion is greater than the downward motion. The key to a successful continuous improvement program is to initiate upward movements and control the downward movements.

Waves Dissected: Each wave has four phases:

- Build
- Crest
- Descent
- Trough

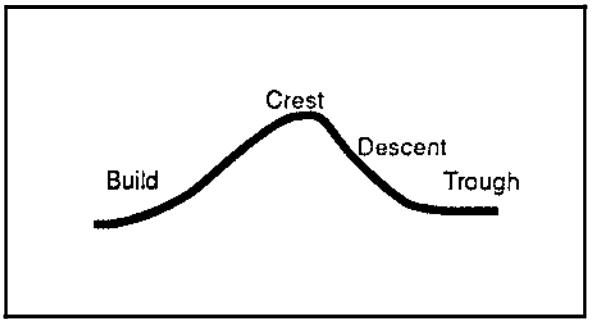


Figure 2. Dissected Wave

The ***build phase*** can be initiated from a positive force or a negative force. A positive force might be one or more persons attending a class or conference and getting energized by discovering some new technology which they believe will improve the quality of their work. An example of a negative force might be a group of engineers all running into the same problem and becoming frustrated with the current way of doing things. Both scenarios result in either champions or teams coming forward (or being volunteered) to make a change. New procedures, tools, or technologies are identified and discussed. As momentum builds an unfreezing of the organization occurs and change can be introduced. Enthusiasm, energy, and expectations are typically high as the build transitions to the crest phase.

The ***crest phase*** is when change is actually introduced and typically tried out on a pilot project. The height of the crest corresponds to the amount of change that was attempted in the organization and to some degree the level of energy in the group that implemented the change. This is where the "rubber meets the road" and new ideas which seem great in theory do not always work quite the way they were expected. This leads to the descent phase.

During the ***descent phase*** some adjustments may need to be made to the change in order for it to be beneficial to the group. The change agents may become discouraged that their change did not go exactly as planned. The key is to not let people get so discouraged that they give up. This is when people need to stick it out and adapt and adjust the new procedure, tool, or technology so that it can work in their particular environment. During the descent, enthusiasm and energy are low and discouragement with the changes may be evident. People need to be reminded and shown what change did occur even if it was not as much as they had hoped. They should be encouraged to celebrate the small successes and appreciate that things are better than they were before.

The final phase is the ***trough***. If everything has gone fairly well, the current trough is slightly higher than the trough before it meaning that some degree of change and improved quality was added to the group's culture. This is the settling in or practice period which must elapse before the next push for improvement. The trough period may be longer if the recent change was particularly difficult to implement or shorter if the change went so well that people are eager to move on to the next challenge. The trough can be used as a re-energizing period before moving on to the next change.

The ideal wave model would show a series of wave crests with an upward trend corresponding to a quality improvement program. The troughs of the waves would also be on an upward trend with the area underneath the troughs corresponding to the amount of new quality culture which has been adopted and integrated into the group. Figure 1 is an example of a good Quality Wave Model which shows a steady, continuous improvement trend. Layers of new, improved quality culture are being laid down in a regular fashion so that continuous improvement is an expected part of the culture.

Bad Waves: The waves shown in figure 3 are examples of "bad waves". Example 1 has a crest period which is too long. This indicates that the change is de-

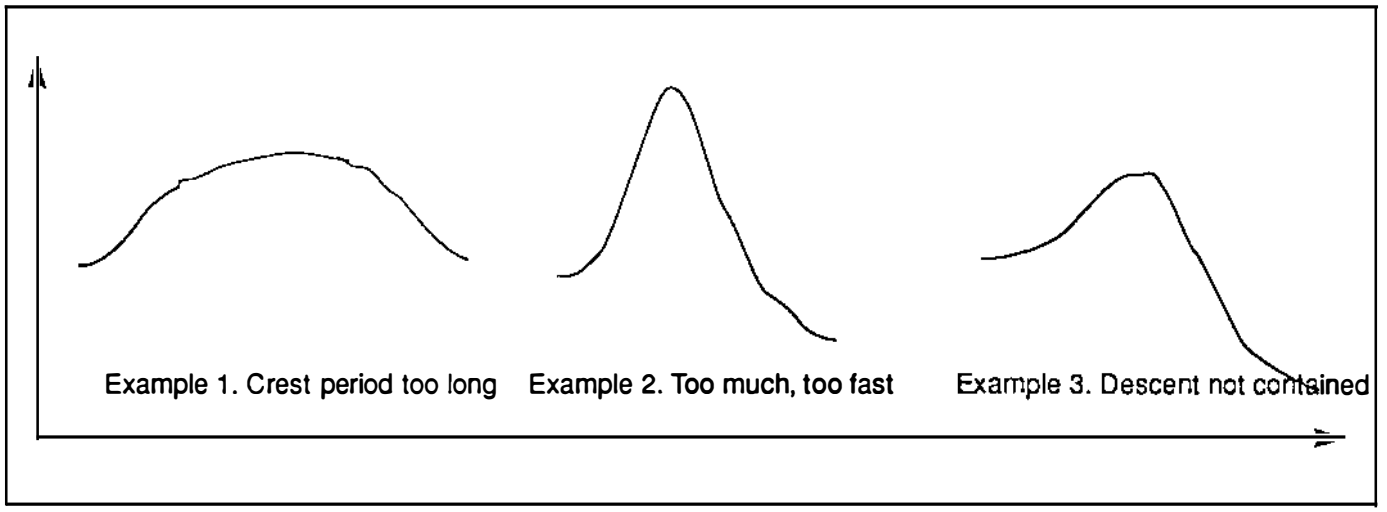


Figure 3. Examples of Bad Waves

pendant upon the champion(s) and is not being added to the overall culture of the group. This can potentially burn out the champion(s) unless they can figure out a way to bring in the rest of the group to adopt the change. This may also be pointing out a problem with the change, it may not be appropriate for the group or the group may not be at a stage to accept this kind of change.

Example 2 shows a build which is climbing too high and too fast. This indicates that the champion(s) may be trying to implement too many changes in a short amount of time. The net result may be that the champions and the rest of the group could become disillusioned or overwhelmed with the magnitude of changes that are needed and abandon the quality improvement program altogether.

Example 3 shows a wave with a realistic level of change introduced in a realistic amount of time. However, because of unrealistic expectations, the champion(s) abandoned the program instead of recognizing that some improvement is better than no improvement. This is a situation where someone must step in and help the champion(s) recognize the success achieved even if it did not live up to their original expectations.

Bad waves are indicators of unusual behavior patterns. They should be treated as "red flags". That is, the quality staff should take notice and investigate to see if there really is a problem. Although Example 3 is almost a sure sign of trouble, a long crest phase (Example 1) may actually indicate that the group has readily and easily accepted the change.

Factors which affect waves: There are several factors which affect the Quality Wave Model and its corresponding levels of changes. These include the people, the size of the organization, its rate of growth, commitment, group culture, and the perception of the quality effort.

People: In a software environment the people can be grouped into engineers, quality staff, and management. Each group has a different perspective on the current state of affairs; it may be wonderful (and energizing), horrible (and de-motivating), or somewhere in-between.

The level of change that can be successfully introduced in an organization is in direct proportion to the energy level and commitment of these three groups. If all three groups are highly enthusiastic and optimistic about the quality program, more of it can be introduced and adopted in a shorter period of time.

Size: The larger the organization the more difficult it is to coordinate activities. In a large organization it is better to select smaller pilot projects and let the wave model act as a ripple effect to eventually pull in other projects. It may also be easier to justify having a full-time Software Quality Assurance engineer or department dedicated full-time to improving quality. In smaller organizations (less that 50 engineers) it is typical to have a core of people dedicated to implementing the quality program on a part-time basis. It may be more difficult to justify putting a software engineer or team full time on a quality improvement program.

Rate of growth: If the rate of growth is rapid then it becomes very difficult to bring people up to speed on quality activities. In our own experience, we doubled in growth in one year and had almost as rapid growth the next year. It made it difficult to move ahead in our qual-

ity program as we spent so much time training people on the existing process. This period was definitely a trough for us as we struggled to just keep up with the demands of bringing in simple tools such as estimation and scheduling and continue to encourage people to use the existing tools we had i.e., IDE Software through Pictures, Frame, Configuration Management Tools, Project Plans, etc.

Commitment: Commitment is important at the individual, the group, and corporate level. Since the individual engineers are the ultimate users, it is very important that they help implement and commit to the quality program. Without their support, there will be no program. If there is no management commitment, creating and maintaining a quality program is almost impossible. The development of such a program is much quicker if people, training, equipment, and tools are provided. The corporate view sees the company as a whole and does not always account for the individualism of each group. When introducing new ideas, the corporate view must consider the attitude that the groups have. To get the groups to work as a whole and form a synergetic company, the corporation must provide strong commitment and support of the quality program.

Perception: What is perceived as quite useful to the quality staff and management may not be widely accepted by the engineers, especially initially. Frequently the quality program is perceived as adding levels of bureaucracy to development and interfering with the creative process involved with producing software. This is especially true if new reporting responsibilities are added to the software development process (e.g., formal reviews, change control, and source code control). It is not surprising that these two have a strong relationship with the morale of the organization. A perception of increased usefulness and decreased bureaucracy will spur enthusiasm while the opposite perception can trigger apathy and indifference.

Even the up and down motion in Figure 1 is not a true representation of the quality program's development. The quality staff see all the little peaks of each wave (figure 2). These are new ideas, problems, and the other factors that can affect the continuity of the wave. It is the quality staff's responsibility to normalize the curve i.e., buffer and filter the changes to provide smooth continuity for the engineering environment.

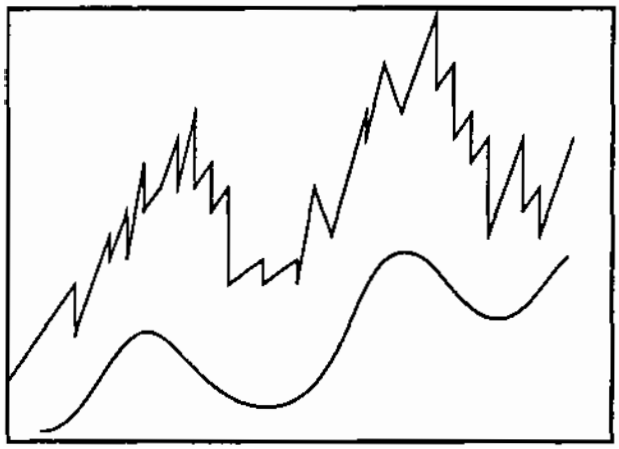


Figure 4. Actual versus Filtered Wave Patter

Culture: Sometimes it is very difficult to overcome the residing culture in an organization. "But, this is the way that we have always done it" may be the response when change is suggested. Therefore, the quality staff must "sell" the new ideas and show the organization how they can benefit from the changes. This sales job occurs at all personnel levels from the engineers to top management. Many times the reasons for one group to adopt change will differ greatly from another group. After a technique or process has been introduced, it must be accepted, practiced, and adopted by the users as a part of their culture.

How to Make Waves: As illustrated in Figure 1 the objective is an overall trend of continuous improvement; however, for every gain there is a loss, and it is important to prevent the activity from dipping below the level of the previous gain. Each wave represents a step in creating a Quality software program. In the appendix is our roadmap for quality activities (wave makers). It is offered as a reference and may be tailored to meet the needs of a group wishing to establish a quality program.

There are two basic ways to start the wave going: top-down, and bottom-up. Figure 5 illustrates these directions of change. The right flow represents the top-down pattern where directives flow from the highest level position of power downward to the individuals. The left flow represents the bottom-up pattern where individuals start making changes and request support from the upper levels of management. Paul Hersey and Ken Blanchard propose that knowledge changes (education) are the easiest to make and group changes are the most difficult. [1]The ideal situation is to have change coming from both directions since this will allow the fastest changes.

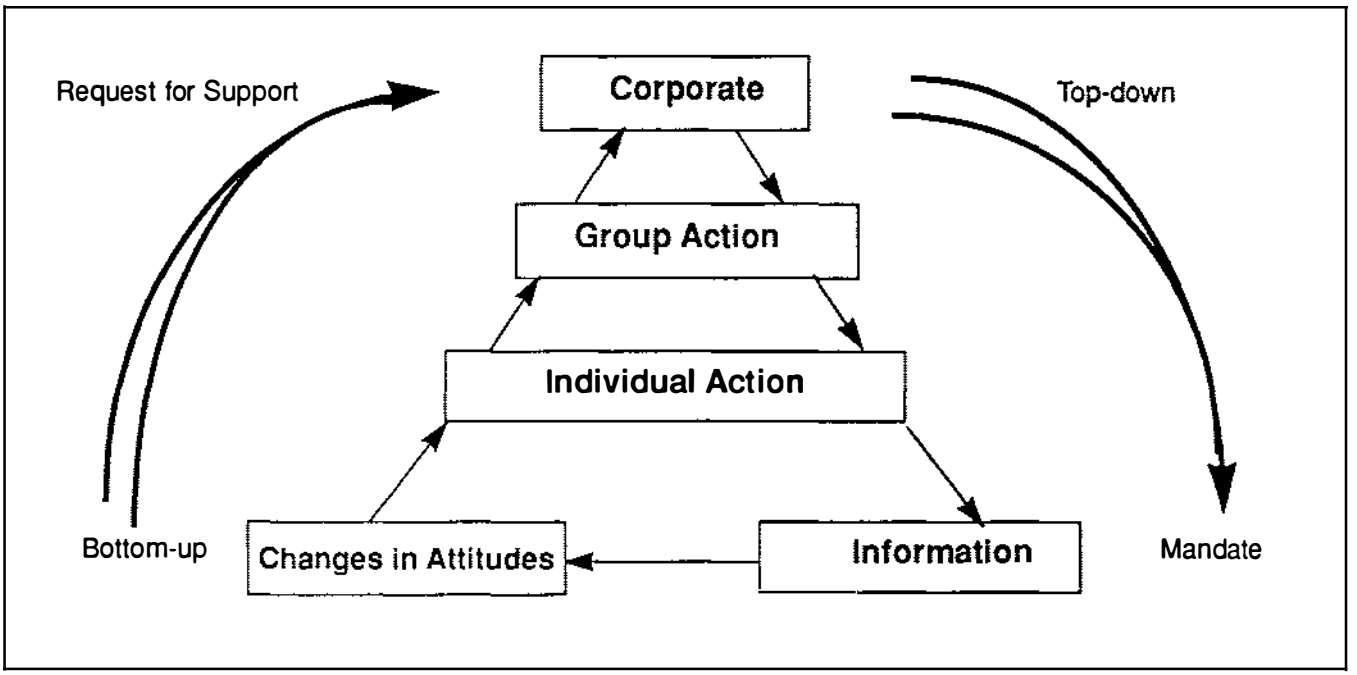


Figure 5. Change Cycles

Top-down: In this approach a company initiative is started with support from the CEO down. It is very important to make a big splash at initiation because by the time is gets down to the engineers, it will just be a ripple. The change starts with a directive for some type of group or organizational behavior: as the engineer becomes more knowledgable, the engineer will either adopt the change, throw away the change if they feel it does not make sense for them or their environment, or modify the change so it can work for them.

Bottom-Up: In this approach the engineers start the quality initiative and pressure their management to support their efforts. As they gain new knowledge about improving quality and how it can directly affect their work, they take it upon themselves to make the extra effort to prove the benefits of the quality program. This type of change begins with the engineer gaining new knowledge, which causes a change in attitude and individual behavior and eventually a change in the group behavior. Basically the engineer becomes the champion or evangelist for the change.

How to Avoid Drowning: Too much change too fast can be overwhelming to everyone concerned. It is very important that people are prepared for changes properly. This is typically done through training and communication. Even something as simple as a "brown bag" training luncheon on how to use a change control tool can greatly help relieve the pressure on engineers to change their work habits.

To follow up on the training and help with individual problems, the quality staff should provide consulting. This means that the quality staff is the resident expert for one on one help. Additional support is established by creating (via training and recruiting) experts and project team champions. This provides more help, inspires more confidence from the engineers, and relieves the quality staff.

New ideas should be introduced incrementally and in small achievable goals. Direction should be set with a roadmap and objectives that keep the department at the forefront of industry. It is extremely important that this roadmap have milestones that are achievable. Make the milestones small enough so that a continued progression of achievement activity can be demonstrated. Time must be scheduled to achieve the goals. (This commitment on the part os the management will also help boost morale.) Make the roadmap visible to everyone e.g., post it on a bulletin board or present it in departmental meetings. The quality staff must also constantly review the program and its progress.

Providing information and experiences from other groups who are developing a quality program will provide insight and new ideas to manage change. It also lets the staff know that they are not the only ones struggling with changes.

Finally small successes as well as large successes should be celebrated.

How to Avoid Drought: There is always the danger that the lull periods may proceed indefinitely i.e., a new wave will not be started and thus the improvement process is halted. This can occur when the downward motion influences are particularly demoralizing and/or when the personnel are overwhelmed (i.e., drowning). It becomes very difficult to generate new enthusiasm. In the earlier wave actions there is always the chance that the down side activities can actually make the quality program regress to a point of less maturity or more chaos and not be able to restart the quality activities.

Fortunately there are several things that can be done to break the lull and start the next wave. One of the most important things to do is to make management support more visible. The quality program must be able to show tangible benefits to the engineers and in order to "sell" the program and boost morale.

The quality staff or champion is also responsible for detecting problem areas and making small changes/ corrections as they occur.

Other wave builders include:

- Setting goals
- Formal assessments
- Education/training
- Recognition for achievements/Rewarding success
- Accountability/Ownership of a change, empowerment
- Tools
- Encouraging new ideas and risk taking
- Enthusiastic champions
- Senior/key engineer commitment/Supporting change

Additionally, a little controlled competition between groups can be very productive and fun. This is also a good response to increased pressure from corporate and customers to improve quality. However, both competition and pressure can have a devastating effect is overdone or allowed to get out of control. Thus, the quality staff must use these tactics judiciously.

If a trough extends too long, it is acceptable (and even recommended) to start over. that is, the quality staff should take a new approach to improving the quality program. For example, change life-cycle model and redesign documentation, customize procedures to be more compatible with the abilities of the staff.

Future: Since this model is relatively new, there has not been enough time to collect sufficient data to provide quantitative values for predicting crest levels, wave lengths, and time periods. At this time, the only way to determine that a crest phase is too long is to study personnel behavior to determine who is supporting quality activities (see Figure 3. Examples of Bad Waves). The same is true for long trough phases. If the engineers are bored or complacent, and/or no improvement is occurring, it is time to start a new wave before the group starts to regress. Future plans include collecting data to provide more quantitative information for implementing plans. The model will also be used in a larger group.

Conclusion: The most important aspect in any program is flexibility. The program should be based on what is most beneficial to the group and its needs. Every group is different and should be prepared for the ups and downs both emotionally and technically. The quality wave model is a way of understanding which will help quality staff and management create a plan for a quality program that includes milestones that are achievable. This plan provides a focal point for the software group and helps the quality staff maintain its vision.

The key to a successful program is to initiate lots of upward movements and control the downward movements, making sure that the activities are visible to the group. During lull periods, people tend to forget what is going on in the program. The quality staff and management must constantly promote and advertise the benefits of a quality program. With time and increases in maturity, continuously improving quality is the accepted culture of the group.

APPENDIX
Our Experiences

How to make waves is based directly on our own experiences of actively working to improve the quality of the software developed in our department. This effort has been going on since 1988. The following wave action is what occurred (and is continuing to occur) in the RISC Software Department.

The Initial Ripple: In Motorola there has been an announcement from the CEO that we will have six sigma quality. At the same time our department started a quality initiative through the efforts of one engineer.

The initial ripple started in an organization of ~20 engineers with basically no formal process for software development. Software Engineers did not even have any standard or consistent way to control source code versions. The initial ripple started with one enlightened engineer and a manager who was looking for a better way to develop software. The engineer used her project as an example by writing requirements and design documents (based on the IEEE standards), having formal reviews, using the UNIX SCCS source code control, and writing and executing test software. This project demonstrated some of the fundamentals of software quality and opened the door for more improvement.

Key elements in the initial ripple were:

- An engineer knowledgable in Software Engineering practices
- A manager willing to try a different approach
- A project to use as a pilot

Wave One: The next wave came with a reorganization of the software department. The department had a new manager who divided the department into three functional groups:

- Development
- Test and Quality Assurance
- Product Support

The group leaders realized that if a product was to move smoothly through the department a policy was needed to describe how the different group's would communicate. This was the beginning of *tactical meetings*. The purpose of the tactical meeting was for the group leaders to discuss ways to improve software quality and the development process. One of the first tasks that the *tactical team* took on was to find out how we could attain Level 2 software maturity by the end of one year. We did an informal SEI assessment and quickly realized that it would take at least 2 years to achieve this goal. Two members of the tactical team committed a considerable amount of time producing what was known as the Life Cycle Document (LCD). The LCD started life as a communication guide for the departmental groups but rapidly mushroomed into a definition of how a product would be developed.

The LCD consisted of two basic components; a dataflow diagram and documentation templates. The dataflow diagram showed who was responsible for producing project documents and which documents were dependent on each other. The documentation templates consisted of IEEE software standards and internal documentation standards. The documents mandated by the LCD were:

- SRS (Software Requirements Specification)
- SDD (Software Design Description)
- SPMP (Software Project Management Plan)
- STP (Software Test Plan)
- Business Plan
- Customer Support Plan
- Documentation Plan.

The LCD was bound and distributed at a departmental meeting. Two of the group leaders presented the document and described how it should be used. A large project was starting up so this was a perfect time to try out the new procedure. The group leaders sat back and waited for the miracles to happen as they and the engineers applied the new guidelines.

As things turned out we had committed our first mistake; we had *defined an ideal* process instead of studying and *capturing the real* process. A month or so went by, with people in the department furiously generating all the appropriate IEEE and internal documents for the new project.

As the documents were nearing completion it became

apparent that we were not really using the process described by the LCD. In fact most of the engineers were surprised to find it on their book shelf when their manager pointed it out to them. In addition there was a sense that we were generating too much paper since there was a significant amount of redundancy in the various documents. The document templates were so vague that the engineers were writing anything just to fill in the sections.

Eventually with some arguing the Life Cycle Document was thrown out.

Key elements in wave one were:

- Reorganizing the department
- Forming the tactical team
- Creating the LCD

In analyzing wave one the build here consisted of the reorganization of the department and the forming of the tactical team. The crest phase was the activity of creating the LCD and applying it to an actual project. The descent occurred as we realized that we had tried to emulate the ideal process instead of the real process. The trough phase lasted until we started to actively capture the real process.

Wave Two: The initial phase of wave two was spent trying to figure out what went wrong. We finally identified the problems as being:

- It specified too many documents for a department of our size.
- No one had been trained in the process.
- It was based on an ideal world not reality.
- It was overspecified with no flexibility.
- There was no mechanism for changing the document.
- It needed more input from people closer to the real work
- There was no formal review mechanism
- We were trying to implement too many things at once.

After identifying the problems with the original life cycle the tactical team formed a new committee, consisting of both technical and managerial staff.

As we began struggling with capturing our process we encountered debates over philosophy, methodology and implementation. A major point of disagreement involved the definition of what a process should encompass. Some committee members felt the need to have a super process flow, encompassing all IEEE documents and internal documents, a detailed handbook, and a complete set of documentation for each project. Other members were at the other end of the spectrum, wanting a minimal process definition with limited documentation. The remaining members were somewhere in the middle. After many heated battles we decided to concentrate on improving two specific areas:

1. Reduce the amount of paperwork and redundancy by simplifying and combining documents.

2. Establish a procedure for reviewing the documents.

In addition we decided to concentrate on the front end of the process, namely the requirements phase and the design phase. We would experiment with different approaches and work them into our culture before we adopted them.

As we began writing and reviewing documents the frustration level began to build. None of the requirements documents were passing the first reviews. The engineering staff was ready to begin doing *real work* (coding) and management was getting concerned about schedules slipping. It became apparent that the committee assigned to the task of developing the process flow was suffering from burnout. The committee had to read all variations of the requirements documents, meet to discuss the current status, offer recommendations and attend all of the reviews, we just didn't have enough time.

Key elements in wave two were:

- Identifying problems with our previous solution
- Narrowing the scope of the problem

In analyzing wave two the build here consisted of understanding what had gone wrong. The crest phase was

Wave Three: At this point the department manager requested that the committee take a step back and prioritize what we were doing. We decided that we needed to split up the work with one committee member assigned to each project. The committee member, project manager, and project engineer would work together on the requirements document. When these

three people involved determined that the document was complete a technical review would be set up. Initially we had tried to apply the process to all of our projects but we realized that we did not have the manpower to do this. We defined a criteria to determine which projects would be the guinea pigs. The criteria we selected were:

- The software had to be going to customers as a product
- The project had to be in the beginning phases.

We identified three projects which met the above criteria. These projects ranged from 'simple': enhancing existing software in a familiar environment to 'complex': developing new software which is tightly coupled with the hardware.

Implementation brought on more disagreements and dissent. Although most department members understand the value of defining how we should build software this was the first time it directly affected their lives. Many technical team members viewed the review-documentation cycle as added work. Not only was it difficult to be developing a process flow in the mist of daily schedules and deadlines, but we also had to convince our technical teams that this was to their advantage as we hoped to get all of the bugs out in the beginning phases of development. Some of the dissention points encountered were:

- The process flow required too much paper generation. The work would progress much faster in the current environment.
- The procedures appeared like another wild management, bureaucratic requirement, that was not deemed necessary by our technical teams.
- Many technical teams did not have the time and resources to invest in developing new procedures.

Fortunately the committee had the support of the department manager and had the luxury of working with each of the teams to the point of hand holding.

Key elements in wave three were:

- Outside intervention
- Working closely with the pilot projects

The build in this phase was the department manager giving us a kick in the rear to get us out of the previous trough. The crest was when we began working closely with each project to walk it through the process. The descent cam as we realized that things were not going as smoothly as we had hoped and as implementors viewed the process as unnecessary bureaucracy. We hit the trough with some successes such as our formal review procedure, this has been part of our culture ever since. However we were to the point of burnout since this was clearly not a 'part-time' job.

Wave Four: The next wave began with the official staffing of a person dedicated to improving software quality.

The quality staff created a roadmap starting with small successes which gather support and then move on to more aggressive challenges which require more support and resources.

Additionally the quality staff took ownership of the quality program.Ownership is very important in that it gives a focal point rather than some nebulous activity that no one knows or cares about.

A quality champion was hired who as a member of the engineering group integrated the quality function into the group instead of having a quality group separate from the design group with a different reporting structure.

Training was introduced on quality issues as well as some tools. Another informal SEI assessment was conducted to get a better evaluation of the department's program. With a few small successes (specifically installation of a configuration management tool, a bug database and collection process, requirements and design reviews) the department was starting to get recognition from outside the department. This boosted morale as did the ever increasing realization that the CEO initiative is sincere.

Since the management support consisted mostly of a statement that 'quality is important and we are going to do it' without a clear and visible plan of how to obtain higher quality, the management support was perceived as "lip service". Coupled with the realization on everyone's part that the quality program is not a silver bullet and unmet expectations, the program began to crumble. There was an ever growing difference of market driven versus quality driven issues. The department was also growing at an enormous rate. Therefore the number of quality staff to number of engineering staff ratio was too unbalanced which did not provide sufficient time for training and consulting. And, the very

improvement that the department had made was bringing too much external visibility too fast which did not allow for the downward motion. Also, the visibility generated jealously in other departments which brought forth a bombardment of ridicule on the department which really lowered morale.

By this time the program was dangerously close to falling apart. A strong effort on the part of the quality staff and management was planned to start the next wave

Key elements in wave four were:

- Managements commitment

The build here was the recognition that we needed a full-time person devoted to software quality issues. The crest came as the quality champion conducted classes, seminars and informal assessments for the department. The descent came as we realized that even a full-time person could either concentrate on a very narrow area and make large contributions or focus on the entire department which would minimize the impact of one person. The trough period consisted of trying to figure out how the quality champion could make the biggest impact. This led us to wave five and looking at tools to automate our process.

Wave Five: Wave five started as we purchased tools to help automate our process. We had made the decision not to purchase tools until we felt that our software development process was part of our expected culture.

New tools were introduced to help specific areas (estimation, reverse engineering, and metrics information systems). Training forthe new tools was either brought in from outside or from internal experts.

More quality staff were assigned to specific, very visible projects with the mandate to be a success.

With more knowledge came the realization that SEI, waterfall, etc. do not work for all projects. And of course, there still existed the overhead impact on schedule and the perception of expanding bureaucracy.

The additional staff used this time to develop and implement their projects which included installation and training.

Key elements in wave five were:

- Tools and automating the process

The build for this wave was the decision to purchase tools and the evaluation and selection period. The crest was the actual implementation and application of the tools. The descent was when it became apparent that some of the tools were not helping solve the problems in fact they were causing more headaches. Our current phase is in this trough. We are practising the tools that are working and to some extent fighting with others. This trough is evolving from a practice and experimentation phase to becoming 'this is the way we do business' culture.

Infinite ripples: We propose that by the time the department has achieved a high level of software maturity, the wave motion is more like a series of infinite ripples. The group assesses the state of the program (which is now the culture), detects areas for improvement, and makes the necessary changes. Only catastrophic changes can cause waves at this point.

Table 1: RISC Quality Program Roadmap

	Jan 92	Apr 92	Jul 92	Oct 92	Jan 93	Apr 93	Jul 93	Oct 93	Jan 94
Configuration Mgmt		Introduction	Training	Policy	Plans	Metrics			
		20% of projects		40% of projects		60% of projects	100% of projects		
				Install hardware					
Estimation		Introduction			Establish database	Training/Tool extend model			
					100% of prelim. estimates				
				Retro-estimate for a baseline		30% of projects		60% of projects	100% of projects
					Collect effort (via time reports)			Revise model	
Metrics	Introduction	Establish database	Metrics generator tool	Training					
		30% of projects		60% of projects		100% of projects			
		40% of metrics		75% of metrics		100% of metrics			
Required Training		present process to new-hires		Draft process training for new-hires		Review training		Incorporate into process section	
					Select required training for managers[a]		deploy 50% of training		deploy 75% of training
					Select required training for engineers[b]		deploy 30% of training		deploy 60% of training[c]
Methods	Structured analysis & design	StP		Define modification method		Modification analysis	Modification design	Document and train	
					Object-oriented analysis		Document & train		
					Object-oriented design		Document & train		
Process Manual		Revise		Version 2	Review, Update, Distribute		Update and distribute every 6 months		
Postmortem	Define	Draft		Add to process manual	Simulator	Compiler	Tools/Tests		100% of projects

Table 1: RISC Quality Program Roadmap

	Jan 92	Apr 92	Jul 92	Oct 92	Jan 93	Apr 93	Jul 93	Oct 93	Jan 94
Aware-ness & Communica-tion[d]	SPT	CIS	SCM	Readings	CIS-special topics				
		SAC	MU council	SW Tech-nical Day					

[a] e.g., Software Project Management; Managing Difficult People; Leadership Institute; White males, minorities, and females

[b] e.g., DQS equivalent; test; methodology; process; requirements

[c] 100% of training by summer 1994

[d] These items are on-going

REFERENCES

1. HERSEY, PAUL and BLANCHARD, KENNETH H. Management of Organization Behavior, Englewood Cliffs, NJ: Prentice-Hall, 1993.

Mutation Testing with the Grail system

I.M.M. Duncan & D.J. Robson

Computer Science, School of Engineering and Computer Science
University of Durham
Science Laboratories, South Road
Durham, England DH1 3LE

email : I.M.M.Duncan@durham.ac.uk
Dave.Robson@durham.ac.uk

telephone : I.M.M.Duncan +44 91 374 3650
D.J.Robson +44 91 374 2635

Abstract :
Some of the problems of testing large scale code are outlined and a prototype testing tool, designed to address some of the issues, is described. A simple example of the use of the tool is given as is a description of the problems met when testing code of more than one thousand lines. Using Mutation Analysis, an attempt is made to identify the most pernicious code mutations, those that remain live longer than other component alterations. Identification of problematic sections of code is made easier using the control flow graph of the source code.

Keywords/Phrases :
Mutation Analysis - Impact Analysis - C Code - Control Flow

Brief Biographies :
D.J. Robson is Director of Studies in the Computer Science Department, University of Durham, U.K.. His research interests are in testing and maintenance and he is a founder member of the Centre for Software Maintenance which was established at the University of Durham in 1986. He is a fellow of the British Computer Society, a Chartered Engineer and a member of the IEEE Computer Society.

I.M.M. Duncan was a doctoral student sponsored by the Science and Engineering Research Council, U.K., and British Telecommunications Research Laboratories. She was recently a temporary lecturer in Computer Science at Durham University and was previously an Adviser in the Durham University Computer Centre with special duties for compiler installation and language teaching. She is currently writing her thesis.

1 Introduction

Program testing naturally occurs at the end of the software development phase and is still the prevalent verification and validation technique. In modern systems, there can be thousands of lines of source code, logically divided into subtasks. Some of the program units or modules may come from old, usable systems. These routines may be undocumented or without specifications. A new system does not necessarily comprise a program cleanly developed from specifications and the code testing must be rigorous enough to verify the product, its constituent parts and their interactions. Integration testing on newly inserted routines, their parameters and their effect on subsequently executed code must be done systematically. Once faults are detected, decisions must be made either to look for faults of a similar nature or, to rigorously test the routine at fault, or both if resources allow. When analysing large scale code, the tester must have the ability to direct the investigation to particular routines or components. It would be impossible to do a full analysis on source code and prior routine testing should be noted for reference. Reduced testing time must not imply cursory inspection; fault removal and increased confidence should be ensured. Testing must also now cover portability issues. Machine and compiler upgrades now occur reasonably regularly and as such, the tester must be aware of any future environment changes. From this, a test technique is required which can test individual units, the interactions between them and simulate environmental changes.

The following sections illustrate a method of testing large commercial and industrial sized programs. The test must be directed efficiently, uncovering the maximum errors per application. The most common problems for the source language concerned must be examined early in the test so as to increase the expectation of early error detection.

2 Mutation Analysis

One of the strictest code examination techniques currently available is Mutation Analysis (MA). It was first developed in the late 1970s. The initial theory was suggested by Hamlet [10, 11]. The work that followed by Acree et al, [1, 2, 3], formalised the technique and named it. As a testing method, MA allows the options of examining code alterations locally and globally. Statement update, and its impact on subsequent code, can be viewed at some strategic local point or at the termination of execution. This paper is primarily concerned with Strong MA, in which the output is examined after execution of the whole code. Weak MA is the analysis of program components to determine if the component has a different value or outcome on at least one test case.

The technique assumes an executable program with associated test cases. The tester alters components in the source code, simulating real programmer errors, and compares the new output to the original on the given test cases. For example, in C, the statement

'while (count <= end-condition)'

may be changed, or mutated to,

'while (count < end-condition)'

If the outputs from the test data differ, the 'mutant' program is deemed dead, i.e. failed, and removed from further analysis. A live mutant program is one which results in identical output

to the original. By introducing the mutations, which are perturbations to the code that form likely faults, and then 'killing' them, the tester can generate a comprehensive test suite. By so doing, the tester is directed towards faults in the code or logic. Redundant components and unreachable code will always deliver live mutations and should become obvious after analysing live mutations. There is an assumption that the code is already nearly correct by the time it undergoes MA. That is, the program differs from its correct version by single component changes. This is the Competent Programmer Hypothesis [6]. A radically incorrect program should have been detected during its development phase. MA aids the development of test data. A central goal of MA is to determine when a system has been adequately tested [6]. The notion of adequacy refers to the ability of a test suite to differentiate between the test code and its mutant programs. That is, the test code behaves correctly on the test suite and all incorrect variants of the code behave incorrectly.

A mutation can simulate faults which are machine or environment dependent. MA requires a series of test inputs, however generated, for which there are known, expected outputs. A Mutant Analyser will change components in the code. Any changes in the processing of the inputs may be evident in the altered output. MA incorporates some data and control flow and boundary analysis techniques. For example, an alteration in a relational operator or a conditional statement would be a control flow test. Mutating variable references would aid the detection of some data flow anomalies. Adding or subtracting 1 to variable references would be considered a boundary test. It is also a useful tool for **Impact Analysis**; that is, the effect of any component alteration can be viewed in the output or at the termination of a statement, function or unit [12, 18]. A component altered in a function called at the start of code execution may radically affect data values or the subsequent flow of control. Alternatively, the component change may be masked or made redundant by the subsequent execution. It is important to analyse these effects for aiding the adaptive and corrective maintenance processes.

3 Testing of Large Scale Software

The number of mutations generated from a source program varies with the number of variables and statements [12]. In a large program, the number of mutations generated must be limited due to time and resource constraints. It would not be practical to generate all mutations from a very large program so the choice and position of the mutations generated must be managed. Ramamoorthy's Triangle Program [5, 16], categorises triangles according to the lengths of the sides input (See Figure 1 later in text). The C version of the code comprises 35 lines of code. The Grail mutation tool, described later, can generate 126 mutant programs from alterations to variable identifiers and references. Some 96 mutants are generated from relational, logical, assignment and arithmetic operators. In a large program of over 1400 lines of code, Grail generated 4156 variable mutations and 3120 operator mutations. Other tests such as constant, conditional and pointer mutations will increase the number of mutants further. The larger program also required a larger amount of CPU time in order to complete its processing. Each mutant must also be allocated at least as much CPU time but alarms must be embedded to prevent non terminating loops from processing continually. Each mutant has to be compiled and executed on the associated test cases. As the number of mutants increases it is important to control the test before it becomes unmanageable.

To test large scale code it is required to know

- Where to start.
- What tests to carry out (control flow, data flow, boundary test).
- When to stop : when a predetermined code coverage metric is fulfilled or when the error find rate is below a particular ratio.

MA systems generally apply mutations in a linear fashion from the first line of the source text down to the last [4]. However, applying the mutations based on textual position does not take account of the calling sequence and internal structure of the code units. It is necessary to find a strategy for reducing the number of mutations applied to a program in order to make the test applicable to large programs. A scheme for selecting the optimum mutations from the sub-groups of control, data and boundary faults must be found. That is, finding the components or the code regions most likely to result in live mutants after analysis. Driving a test by focusing on the most likely faults would increase effectiveness for a minimum cost and also indicate which sub-groups of error conditions are more relevant to large scale software. The number of possible program mutations can realistically be reduced to those which are primarily necessary.

A real commercial or industrial program of several thousand lines or more could still take more time to test than is available. There is a necessity to focus on any critical or error prone regions within the code, that is, those parts which have to be tested more thoroughly than others. A more thorough examination can be done, if costs permit, by simply allowing the test to continue through the less critical or error-prone routines. A test route or procedure should be found which will generate the optimal test, for the available test cases, in as short a time as possible. That is, a tester would wish to determine where the remaining live mutations reside in the code and to ascertain that information without resorting to a full code mutation, if possible [7]. The test must be driven at the most efficient rate possible. It is important to initially simulate the common faults in the most crucial routines, however they may be defined. A mechanism is required to direct the mutation application through the source code. The code linear sequence call graph, based on control-flow, is a useful apparatus for this task.

A linear sequence is a maximal group of statements such that if the first statement is executed then so also is the last. Embedding test case coverage probes at the linear sequence level of the code gives more insight into actual code coverage than embedding at the routine level. Linear sequence traversal indicates that all linearly connected statements are traversed and thus gives an indication of total code coverage. A control-flow graph is a diagram of the connections between program regions (statements or linear sequences). Using the definition of a flow graph from Fenton et al [8], a control-flow graph is defined to be a finite digraph G incorporating the distinguished start (source) and stop (sink) nodes. The in-degree of a node is the number of edges entering the node. All nodes, except the source, have an in-degree of one or more. The out-degree of a node is the number of edges leaving the node. All nodes, except the sink, have an out-degree of one or more. The nodes of a control-flow graph are program regions and the edges are the flow of control between those regions. As MA is a structural testing technique, the control-flow graph can be generated from the linear sequence connections. The nodes in the test program's control-flow graph are linear sequences. A node with an out-degree of two is a predicate node, the out bound edges corresponding to the True

and False control paths and the connected nodes being the linear sequences at the start of the relevant code. (See Figure 4 later in text.)

If linear sequences are mapped according to their call sequence and the call graph generated, a mutation within a particular sequence can be viewed through its impact on the following code. A linear sequence higher in the graph, i.e. nearer the root or source, may be executed earlier than one later in the graph, nearer the code termination (sink of the graph). As such, any component changes within it may have greater consequence than a component change in a sequence close to the sink. Alternatively, the component change close to the sink node may be the more likely to create a live mutant. A back edge, such as is formed by a loop, will complicate this topic. Linear sequences within loops are considered positioned by their first call and the loop is not unravelled. The cumulative count of live mutants are plotted against mutations generated. Mutations initially generated in a higher node of the call graph will generate a different plot to mutations initially generated in a later called node. By driving the mutation application via the linear sequence call graph, the tester can gauge whether faults in the earlier called routines, critical or otherwise, are more likely to result in live mutants than faults induced in later called routines. The tester can also isolate which segment, function or unit exhibits the most live mutations and therefore requires more analysis. A zero kill rate for a linear sequence may indicate non traversal or inadequate test data. Untraversed sequences indicate either a problem with the test data or unreachable code. These sequences can be discovered by mapping test case traversal against linear sequences. Using data, control and boundary error mutations, the tester can determine which group generates the greatest number of live mutant programs. From this knowledge a tester can determine problematic constructs and linear sequences within the code. As sequences and functions are altered, the directed MA test lends itself to Revision or Regression testing [7]. Knowing which test cases traverse the altered code allows the tester to reduce the re-application of test cases.

It is therefore necessary not only to examine the effect of any induced alterations but to determine whether driving the test via the call sequence order improved the test efficiency. That is, an improved test is one in which live mutations are found early in the test sequence. This raises several issues. Any improvement made by driving the test via the call sequence may be program or complexity dependent. It would be necessary to conduct many trials of different programs in order to achieve a valid conclusion. Any improvement may also be test case dependent, so it would be necessary to alter the test case order. Some test cases may be considered good for killing mutations because they have a high statement coverage, others because they are special case selectors. The tester must utilise both types of test cases, but the former is better for an initial test in order to remove as many live mutations as early as possible. It is important to discover whether a control flow based test is more likely to uncover error groupings [15] than a linear, or textual, based test. A textual test is one in which mutations are generated line by line from the source code with no regard to program flow. Another test is required to determine the pernicious mutations, that is, the mutation types most likely to remain live. For an efficient test, knowing that most mutants are unstable and die quickly, it is important to simulate faults which require special analysis and test cases. That is, it is necessary to simulate common faults, be they boundary, control flow or data flow groupings. However, these 'primary' mutations may not be applicable to all programs, but to program types determined by size, complexity, coding techniques employed or components used.

From these initial deliberations it was decided to build a prototype mutation testing tool for C source code. The language C was requested by the research sponsors and it was considered

a useful exercise because of the prevalent use of Unix systems and C code in the academic as well as the industrial world. The prototype, now called the Grail, was built at Durham University and was an attempt to discover answers to the aforementioned problems of managing a mutation test on large scale code.

4 The Grail System

The Grail prototype testing tool accepts SUN-OS C source code in single units. The Grail allows the tester to choose between 1 and 11 mutation types; relational, arithmetic, assignment, increment-decrement and logical operators, variable reference and boundary, constant and unary replacement, context negation and simple pointer alteration. These groups simulate programmer errors of data assignment and reference, control flow and boundary errors as well as mutations imitating environmental changes. The user is prompted to choose either a textual, a preorder or an inorder test mechanism. The first of these mechanisms is the usual technique used by MA tools and simply generates the component mutations as it finds them in the source code text. The preorder and inorder tests use the linear sequence call graph of the code to generate the mutations based on their call position. The tool has been in use for several months and has been used to test code of up to a few thousand lines. The Grail has given indications of which mutations give rise to greater numbers of live mutant programs and also indicates which segments or functions require further attention. Part of the system's output is a chart of live mutants per linear sequence or function. This information is used to determine further test data and is useful in indicating problematic functions or linear sequences and constructs. Data regarding which test cases kill any particular mutation is also stored. The tester can analyse this file to determine redundant test cases.

5 Example of Use

The original programs used to test the traversal mechanisms and to determine which mutations were more pernicious were C code translations of programs used in two doctoral theses, [9, 16]. Later, and larger, programs were taken from text books [13, 17], and code available on Unix systems. The first section of the Grail itself, some 2000 lines of source C code was also tested.

The example program in Figure 1 is Ramamoorthy's Triangle program taken from various texts [5, 16]. All linear sequences within the code are traversed by the six test cases shown in Figure 2. These were taken from one of the texts, [5]. The test case to linear sequence traversal is determined from a matrix stored by part of the Grail system. The matrix can be examined to determine untraversed sequences.

The eight relational operators mutated in the example, are in linear sequences 2, 4, 6, 11 and 13. The program is very simple and the preorder traversal is similar to the textual sequence traversal, see Figure 3. The sequences containing the relational operators occur in both traversal mechanisms in exactly the same order. See Figure 4 for the Linear Sequence Call Graph.

Ramamoorthy's Triangle Program
An example of linear sequence connection

```
/* The correct program input is three numerically descending integers, the
   sides of a triangle. The program outputs the triangle type described.
   LS refers to linear sequence number. Some are required for connectivity
   e.g. LS 16, 18, 20
*/
#include <stdio.h>                                          /* LS 1 */

int a,b,c,d ;
main()
{
 scanf("%d%d%d",&a, &b, &c);
 printf("%d\t%d\t%d\n",a,b,c);
 if ( a >= b && b >= c)                                     /* LS 2 */
    {                                                       /* LS 3 */
     if (a == b || b == c)                                  /* LS 4 */
         {                                                  /* LS 5 */
         if ( a == b && b == c)                             /* LS 6 */
             printf("%s\n", "Equilateral");                 /* LS 7 */
         else
             printf("%s\n","Isosceles");                    /* LS 8 */
         }                                                  /* LS 9 */
     else
         { a = a * a;                                       /* LS 10 */
           b = b * b;
           c = c * c;
           d = b + c;
           if ( a != d)                                     /* LS 11 */
              {                                             /* LS 12 */
                if ( a < d )                                /* LS 13 */
                   printf("%s\n","Acute");                  /* LS 14 */
                else
                   printf("%s\n","Obtuse");                 /* LS 15 */
              }                                             /* LS 16 */
           else
               printf("%s\n","Right Angled Triangle");      /* LS 17 */
         }                                                  /* LS 18 */
     }                                                      /* LS 19 */
 else
    printf("%s\n","Triangle Sides not in order");           /* LS 20 */
}
```

Figure 1

Test Cases

TC1	TC2	TC3	TC4	TC5	TC6
2 12 27	5 4 3	26 7 7	19 19 19	14 6 4	24 23 21

Figure 2

Traversal Sequences

```
Textual Traversal : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Preorder Traversal : 1 2 3 4 5 6 7 9 19 21 8 10 11 12 13 14 16 18 15 17 20
Inorder Traversal : 21 19 9 7 6 8 5 4 18 16 14 13 15 12 11 17 10 3 2 20 1
```

Figure 3

Linear Sequence Call Graph

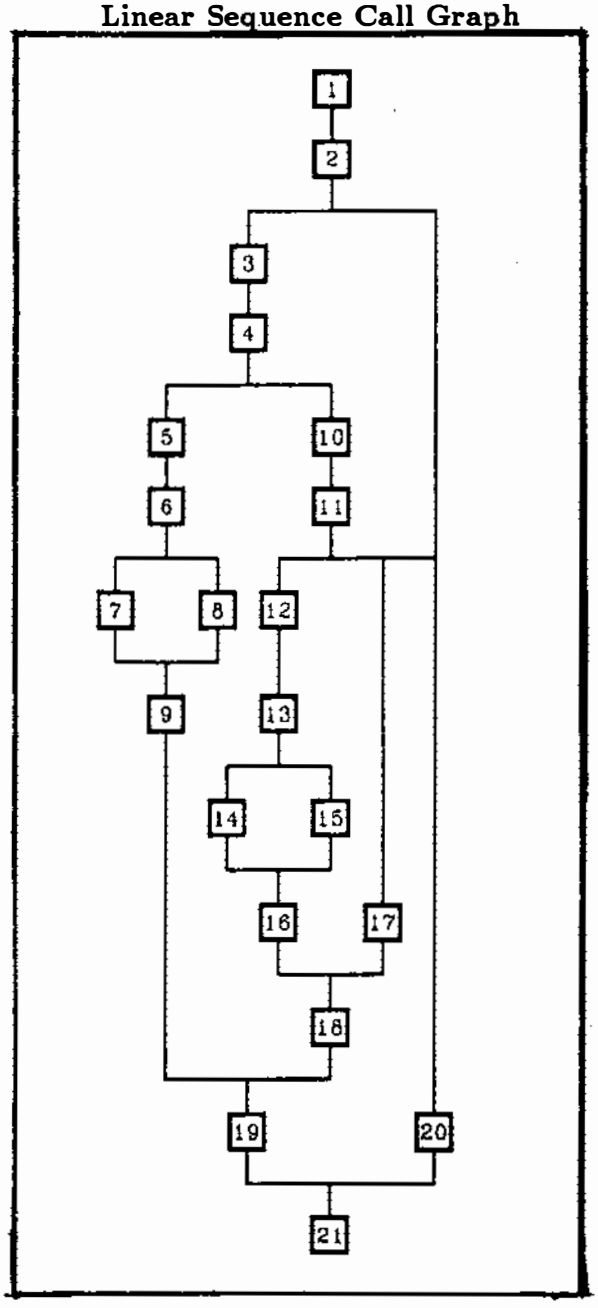


Figure 4

Live Mutation Plots

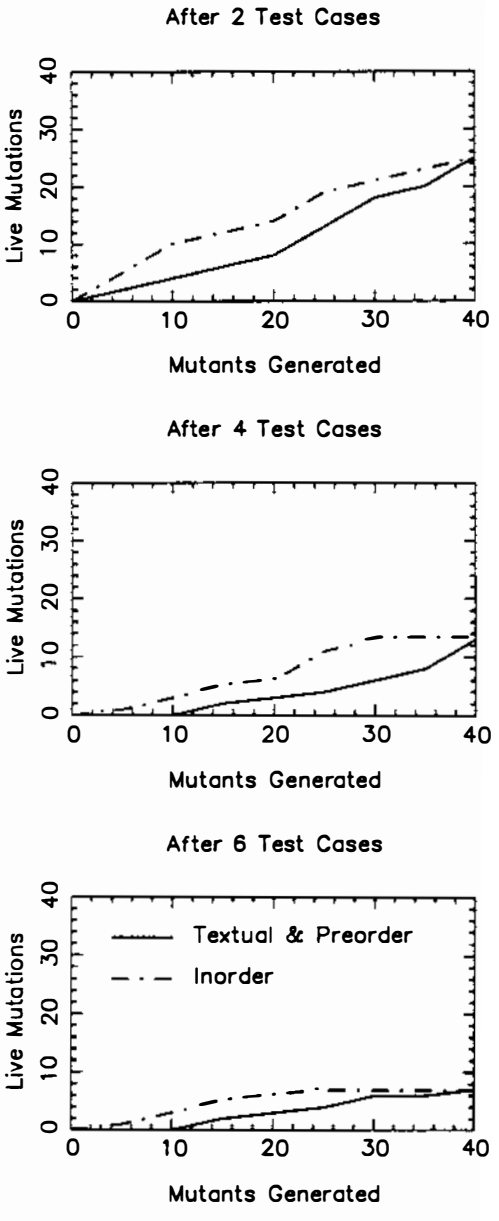


Figure 5

The plots of live mutations found against mutations generated are identical. This is a common occurrence for small programs. Each component is detected in the code and mutated in turn. Each relational operator will be converted into the five other tokens from the relational operator set, $\{==, != , <=, >=, <, >\}$.

The live mutations to mutations generated are then drawn automatically from the tables output from the Grail. The ones shown in Figure 5 are some of the relational operator plots from the cumulative application of test cases 1 through 6. Similar plots can be generated for all the other mutation operators. For Ramamoorthy's Triangle program, all the test cases and mutation operators, with the exception of the Logical operator, exhibited inorder traversal as the best of these mechanisms for mutation generation. The ordering of the test cases was altered to check its effect on the plots generated. There was no apparent change in the foremost mechanism. This implies that mutations formed in linear sequences close to program termination are more likely to remain alive than those generated towards the start of execution. However, Ramamoorthy's triangle is a very simple program with no loops and only one function. It was thought that an increase in code complexity may alter the efficiency of the traversal strategies.

Small programs tended to exhibit inorder traversal as the best technique for all mutation types. In all the early trials the mutation operators that generated the most live mutations per mutation generated always included the relational, the variable reference and the variable boundary mutations. Variable reference mutation operators altered individual variable definitions and references to other variables in scope. Variable boundary mutation operators added or subtracted a constant to the variable references and also formed the positive and negative absolute value of references.

6 Large Scale Testing with the Grail

Testing large scale code with the Grail system revealed that there are a number of interesting factors worthy of further investigation. Programs of up to 2000 lines were tested. The preliminary results from the tests were initially surprising. The small programs, code of less than 100 lines, showed that the textual search and mutate mechanism was the least optimal of the three test routes. The opposite appeared to be true for large code. However, this may not be a true measure of the traversal techniques but may be connected with the coverage of code by individual test cases. As the code increases in size, each individual test case will cover a smaller percentage of linear sequences (statements) than a test case for a small program. As such, the effect of testing via the control structure is nullified due to any one test case taking a particular path through the control flow graph. There is no necessity to mutate components in all parts of the code when only one path is being traversed. If the path taken through the code traverses a right subtree of the control flow graph then the inorder and preorder traversal mechanisms will be the worst case search mechanisms. What is required is a traversal mechanism based on the path taken by a particular test case. At present the Grail holds a coverage matrix of test cases to linear sequences. It is possible to use this matrix to generate mutations only in sequences traversed by the applied test case. The current version of Grail does not support this. However, the output from the Grail has been adapted to show the effect of mutating along the execution path. In the current trials, textual traversal appears to be the best tested route for both the full mutation and the execution path mutation tests. The differential between

the textual and the next best traversal technique is radically reduced when viewing the path only strategy. The differential between the traversal strategies, a measure on the gradient of the live mutations to mutants generated graph, would be expected to reduce when the path only mutations are considered. This is due to the removal of the effects of untraversed linear sequences. From the small scale program tests it appeared that an inorder search and mutate of components was a more efficient strategy for finding live mutations. Large scale tests show that a textual search and mutate order is more likely to generate live mutations earlier in the test. As the textual order is not based on the control structure of the code under test, this implies that the masking or proliferation effects of induced faults are not position dependent. Effects due to the placing of output statements must be taken into account. However, the best strategy may not have been determined and others should be tested.

Large scale code tests were continued to determine the most pernicious mutations. On smaller scale code, the relational operators and the variable reference and boundary mutations always exhibited the greatest ratio of live mutations to mutations generated. On large scale code, the same rule held true. However, other mutations should not be ignored. Depending on the nature of the code, or test cases, pointer and address manipulations and assignment operator mutations have exhibited pernicious mutations. Both groups also give rise to 'Zombie' Mutations. That is, mutations which on one execution are dead, but on another are alive. These were obvious from the Grail tests because each test case was executed on the code for each of the three traversal techniques and for each of the eleven mutation groups. The number of live mutations for each technique did not always match but varied by less than 2%. The assignment operator mutations generated the most Zombie mutations because of the incremental nature of C assignment operators. For example,

'a = 0' is mutated to 'a + = 0' and 'a − = 0' etc.

Depending on the values stored in memory at the time of execution, the mutations may be live or dead. A partial solution is to ensure that all variables are initialised. Mutations of the initialising statements should then be removed from the statistics. In C code testing, a tester must remove equivalent mutations as well as Zombie mutations to achieve a better measure of test data adequacy.

7 Results of Preliminary Trials

Several areas of interest have come to light with Grail testing on small and large scale C source code.

- Traversal Mechanisms. Analysis of the live mutants per mutation generated for each of these mechanisms shows that the best mechanism varies depending on the complexity of the code under test. The best tested mechanism (mutation test route) is normally static for each of the eleven mutation groups throughout the test case application, that is, it does not appear to be test case order dependent.
- Problematic constructs. This is the detection of which mutations generate more live mutants per mutations generated. Although this is program and test case dependent, trials show the relational operators and manipulations on variables to be the most pernicious.

- Zombie Mutations. Mutation of C programs give rise to Zombie Mutations. These are mutant programs which are live or dead depending on memory garbage. e.g. altering '$a = 0$' to '$a+ = 0$'can result in either a live or dead mutant program. These mutations are difficult to detect unless tests are repeated. Memory must be initialised, if possible, to remove Zombies. However, this in itself may mask other problems that occur in the original code. Zombies occurred in less than 2% of the mutations generated in the trials.

- Focused Testing. The Grail system indicates the routines displaying the live mutations. This information is useful for either updating the test data and/or focusing the test on particular problematic routines. If a particular routine or unit exhibits a greater ratio of live mutations in comparison to other regions, then that routine is either not traversed or there may be problematic constructs within it.

- Test Case Kill Rate. The Grail indicates which test cases kill particular mutations. Test cases can then be ordered by their killing potential for future revision or regression tests.

- Integration and Impact Analysis. In a large system it is unlikely that a tester would wish to test the whole code. Functions and units should have undergone unit testing prior to integration tests. An integration test can be performed under an MA harness; parameters can be mutated on entry and exit from the newly inserted code. The new code can be mutated to determine underlying faults by reference to its impact on subsequently executed routines. MA can generated boundary, data assignment and reference and control flow faults to analyse rigorously the execution effects of alterations along the execution path containing the newly inserted code. Any alteration effects can be viewed at strategic points along the execution path. That is, MA can be used as a tool for impact analysis.

- Test Case Coverage Problem. The question is raised as to when a large program or system unit is considered tested.

  - When the individual functions have been traversed at least once? The interconnection problems may not have been addressed and a single traversal of functions is not a sufficient test for parameter checking.
  - When a percentage of linear sequences have been executed? However, this assumes that not all paths or conditions will have been tested.
  - When all the linear sequences have been executed? This is difficult to achieve and is not strict enough for MA.
  - When a percentage of mutants have been killed? The ones not killed could indicate serious defects in the code.

  DeMillo [6] rightly suggests that testing should only stop when the test suite is adequate to distinguish the code from its incorrect mutants. However, as the number of mutants generated is of the order of the number of variables squared, a large scale code test, such as would occur for a system unit test, would require much time and computer resources. Further analysis of large code is required to determine useful guidelines for the stopping criteria.

8 Conclusion

The Grail system has given rise to some serious issues in the testing of large scale code. A study was conducted to discover if the efficiency of a test could be improved be generating mutations along a sequence determined by the control flow graph of the test program. In small programs an inorder component search and mutate mechanism was found to be more efficient in finding live mutations that a preorder or a textual search. However, in large scale code the textual search and mutate order, that is, a source line by line order, was found to be the most efficient of the three strategies. When the mutations were considered along the execution path only, the textual mechanism was still the best of the three mechanisms. This gives rise to the consideration that the effects of the mutations should only be considered along the path from mutation to first output (of affected component). This is effectively a form of Firm Mutation Analysis, [18], in which components are analysed at a logical position prior to termination. Other traversal techniques, such as level-order search and mutate, should also be tested.

More work is required to determine code sequences or constructs that require stricter tests. Assumptions have to be made about prior testing of routines and integrated into an overall strategy. A Mutation Test on a large program of more than a few thousand lines is possible but extremely time consuming unless the vector strategies researched elsewhere are incorporated [14]. For revision and regression tests a more focused approach is necessary. Only the updated routines require analysis with respect to subsequently called routines. Mutation Analysis can be focused onto particular routines and can be made to simulate common programmer errors.

9 Acknowledgments

The author would like to thank the referees for their valid criticisms which improved the original paper.

References

[1] A. T. Acree. *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1980.

[2] A.T. Acree, T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, April 1979.

[3] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven CT, 1980.

[4] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the Mothra Software Testing Environment. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), April 1978.

[6] R.A. DeMillo. Test Adequacy and Program Mutation. In *Procs. 11th International Conference on Software Engineering*, pages 355–356. IEEE Computer Society Press, 1989.

[7] I.M.M. Duncan and D.J.Robson. Parameterized Mutation Testing. *Journal of Software Testing, Verification and Reliability*, 1(4), January- March 1992.

[8] N. E. Fenton, R.W. Whitty, and A.A. Kaposi. A Generalised Mathematical Theory of Structured Programming. *Theoretical Computer Science*, 36:145–171, 1985.

[9] M. R. Girgis. *Studies of Program Test Coverage Criteria and the Development of an Automated Support System.* PhD thesis, Liverpool University, 1986.

[10] R.G. Hamlet. Testing Programs with finite sets of Data. *Computer Journal*, 20(3), 1977.

[11] R.G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, SE-3(4), 1977.

[12] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *Transactions on Software Engineering*, 8(2):371–379, July 1982.

[13] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[14] Aditya P. Mathur and E. W. Krauser. Modeling mutation on a vector processor. In *Proceedings of the International Conference on Software Engineering.* IEEE Computer Society, 1988.

[15] G.J. Myers. *The Art of Software Testing.* Wiley and Sons, 1979.

[16] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. GIT-ICS 88/28.

[17] W.H. Press, B.P.Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1992.

[18] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

Selecting Functional Test Cases for a Class

John D. McGregor[1]
Douglas M. Dyer[2]

Abstract

An important phase in the testing process is the construction of test cases and the selection of coverage strategies to indicate when a sufficient number of appropriate cases have been created. A number of coverage strategies have been proposed in the literature, but they have varying error detection capabilities and require widely varying numbers of test cases. The purpose of this research is to investigate a set of coverage strategies with regard to the object-oriented paradigm, to order the strategies with respect to their error detection capabilities, and to suggest which of these represent "adequate" testing. One of the strategies is examined in detail with respect to the selection of test cases. A technique for selecting the test cases necessary to achieve the desired coverage is also provided.

Biographical Information

Dr. John D. McGregor is an associate professor of computer science at Clemson University. With David A. Sykes, he is author of *Object-Oriented Software Development: Engineering Software for Reuse* published by Van Nostrand Reinhold. He is also author of numerous research and technical papers on object-oriented software practices. He is an international consultant to industry and is active in ACM and IEEE CS.

Douglas M. Dyer is a software developer for Software Technology, Inc. His interests include software engineering and data communications.

[1]Object Technology Group; Department of Computer Science; Clemson University; Clemson, SC 29634-1906; johnmc@cs.clemson.edu
[2]Software Technology, Inc.; 5904 Richmond Hwy.; Alexandria, VA 22303

1 Introduction

Recently the testing of object-oriented software systems has received much attention. Some techniques have been borrowed from the procedural world, adapted, and applied to these systems while new approaches have been developed as well. The separation of specification and implementation practiced in object-oriented software has sharpened the distinction between functional testing, testing based on what a component is supposed to do, and structural testing, testing based on the structure of a component's code.

For functional testing, the goal is to prove that the software performs in conformance with its specification. In order to achieve that goal, test cases are constructed, executed, and the results compared with expected behavior. Coverage strategies describe techniques for creating test cases so that the cases exercise "all" of the specified behaviors of the software where "all" is defined by the coverage strategy. For purposes of this research we will assume that the behavioral specification of a class can be modeled by an approximation to a finite state machine. The items of interest in this type of representation are the states and the transitions between the states. The coverage strategies presented in this paper will be discussed in terms of this finite state representation.

A "state", in the usual sense, may be taken as a vector of values containing one entry for each of the attributes of the object. We will denote one of these value vectors as v. The **state space**, S, of an object is the cartesian product of the value ranges for each of the attributes. For example, if a component has two attributes, A and B, where A is a boolean attribute and B ranges over { -1, 0, 1}, the state space of the component is { (T, -1), (T, 0), (T, 1), (F, -1), (F, 0), (F, 1) }.

From a modeling perspective this is not acceptable since every change in any variable results in a "new" state. This potentially leads to an infinite set of states that is worthless in guiding modeling or testing activities. Therefore, we will consider a "design" state, s, to be some meaningful set of value vectors such that vectors in one set differ from those in another, in some meaningful way. For example, typically the value of a single integer variable, X, changing from 1 to 2 to 3 and on is of only local interest and calling each value a separate state provides no additional information; however, if X changes from being positive to zero or changes from zero to negative, such changes are often viewed as significant. A set of meaningful states for X would be positive, zero, and negative.

In software design we will most often be interested in states defined by an aggregated set of attributes. Therefore, our working definition of state is:

> *a set of value vectors that share some behavioral attribute of interest.*

We will further assume that the specification of a component includes a state space, S, and an event space, E, which together define the behavior of the component. A transition space, T, assists in modeling the dynamics of how objects move from one state to another.

The state space, S, is the set of states for the component. S is assumed to meet the criteria that if v is a possible value vector for the component then this implies that $v \in s_i$; where s_i is one of the states in S. That is, the state space S includes every possible v for the component. The set S is a "cover" for the behavior of the component. Further we assume that, while the grouping of v's into s_i's is by no means unique, for a given decomposition,

the states are disjoint. So S forms an exhaustive, mutually exclusive cover for the set of value vectors or the behaviors of the object.

The event space, E, of a component is simply the set of methods that are invoked in response to messages and that operate on the representation of the component.

The transition space, T, of a component is a set of invocations of the events in the event space. An event may correspond to more than one transition if different sets of input vectors result in the object being placed in different states. A transition can be represented by a set of triples of the form

(<input vector> event <output vector>)

The input vector for a transition includes the current state of the component and any input parameters required by the event. The output vector includes the resulting state of the component as well as any values returned by the event.

It is the hypothesis of this research that an algorithm can be developed for selecting test cases, for functional testing, based on the state machine representation of a class. Further, this method is superior to an axiomatic approach in that it is much easier to define and comply with adequacy criteria than it is to prove a set of axioms is complete. It is possible to develop a test case selection approach that provides much guidance in developing the test cases as opposed to the axiomatic approach which has no algorithmic construction process.

2 Background

The research presented in this paper is based on work from several areas. First is the ability to specify the behavior of a class as a finite state machine. Second, are the requirements and techniques of functional testing and finally the blending of these two areas to test the behavior of the software implementation of state-specified components.

2.1 State machines

A number of representations have been used for object-oriented systems in general and classes in particular. Rumbaugh et al[5] use three separate models to fully represent an object-oriented design. The object model is a modified entity-relationship diagram that captures the static relationships among entities. The functional model is a set of data flow diagrams that represent the processing carried out by the main functions in the system. The third model, the dynamic model, is a set of state diagrams, one for each of the important classes in the system. In fact, many object-oriented analysis and design techniques use a state diagram as one of the representations for class behavior.

A state machine is specified as a set of states, represented by particular combinations of attribute values, and a set of transitions. The transitions may be labeled with a name for the transition and an input and output. For our work with classes the name of a transition will be the name of the event that enables that transition. The input will include not only the values of parameters passed into the method but the value vector for the current state of the object as well. This is actually no addition in most object-oriented languages since a pointer to the current object is an implicit parameter to each method in many object-oriented languages.

2.2 Functional Testing

Functional testing, also referred to as black box or specification-based testing, investigates whether the given implementation behaves as expected based on the specifications. In fact, the test cases can be constructed without access to the implementation. Functional test cases are of the form:

<sequence of actions, expected behavior>

where the sequence of actions is intended to produce the expected behavior or

<sequence of actions, sequence of actions>

where the two sequences are expected to produce the same behavior from the component.

Doong and Frankl[2] used the second form of test case for testing data abstractions. They first construct a set of axioms for each abstraction from its specification. These axioms are then used to produce sequences of actions that are identical, i.e. each sequence leaves the component in the same state, according to the axioms. This approach allows the actual execution of the test cases and certification of the results to be automated. The technique still relies on the human oracle in the production of the axioms and the creation of the equivalent sequences. There are practical problems with this approach including knowing when a "complete" set of axioms have been developed.

3 Classes as state machines

Protocol testing[1] has made use of a finite state machine (FSM) representation for software systems. This representation has also been found to be an effective means of representing the behavior of an instance of a class. First, objects do have state. After the creation of the object, each message either accesses or modifies that state. Second, an object is the target of a stream of messages occurring in a somewhat arbitrary order. This multiple entry point perspective is accommodated by constructing each method so that it may be activated in any state.

We will place certain restrictions on the objects that will be represented as finite state machines. These restrictions include:

- *The behavior of the object can be defined with a finite number of states.*
- *There are a finite number of transitions between the states.*
- *All states are reachable from the initial state.*
- *Each state is uniquely identifiable.*

3.1 The loss of formalism

The work on protocol testing assumed a more rigorous definition of the machines used in its technique than is usually possible for the definition of a class. It assumes that each machine is a model of a finite automaton. In particular, Chow makes the same assumptions as were listed in the previous section, but also assumes that each machine is completely specified.

This last restriction is seldom obtainable for classes in general; however, as Chow states, this restriction can be violated with suitable additions to the methodology.

Finally, protocol testing assumes that each state/transition combination is uniquely identifiable when using its full – input, event, output – label. A careful definition of **input** is required to approximate this restriction. It is not sufficient to use the method name as the input to a transition, because the same method may result in one of several transitions depending upon the values of parameters to the method and the values in the current state. The input must encompass all of these factors – the name of the method, the values of parameters, and the current state value vector – in order to be unique or identifiable. While this, on the surface, violates the definition of a finite automata, consider the unique structure of a class. A class introduces a new level of scope. A method has "global" access to the data defined within the class definition of which it is a part. Thus the **input** to a method includes all of this state information.

Additional design restrictions may be required for class definitions to more closely approximate the finite automata. In particular, each class might be required to have a reset method that would move the object back to its initial state and make the class fully connected. It is already considered good object-oriented design to implement methods so that any method can be invoked when the object is in any state. This results in the object being able to respond to every input in every state.

3.2 State machines and inheritance

An important consideration in the efficient testing of object-oriented systems is the exploitation of the inheritance relationships between definitions. Harrold et al[3] developed the hierarchical incremental testing (HIT) algorithm that supports the reuse of test cases from the parent classes in the testing plan for a child class. The mapping of a state machine for a parent onto that for the child presents the same opportunity to reduce the effort in creating state diagrams for child classes. A consistent mapping of machines along the inheritance hierarchy provides a means for reusing information from one level to another.

McGregor and Dyer[4] present a complete analysis of the relationship between the state diagrams for a class and its subclasses. That analysis assumes a restriction on the use of inheritance defined as follows.

"Strict" inheritance assumes that each subclass contains the public interface of its parents as a subset of its public interface. However, the requirements of strict inheritance go beyond the textual matching of method names; the **intention** of those methods must remain constant. This can be more formally expressed by stating a **method invariant** for each method that is inherited by the new class:

> *The pre-conditions on a method in the subclass may only be weakened relative to the pre-conditions on the same method in the base class. The post-conditions on a method in the subclass may only be strengthened relative to the post-conditions on the same method in the base class.*

This leads to the statement of a **class invariant** as:

The specification of a class must include, as a subset, the specification of each of its base classes. Additionally, the method invariant must hold for every inherited method in the subclass. By implication, every class has in its specification the specification of the root of each inheritance structure within which it participates.

These two requirements produce an inheritance hierarchy in which the subclasses are subtype compatible with their superclasses and polymorphic substitution of subclasses for a class is possible. They also produce a structure about which logical inferences can be made.

The results of McGregor and Dyer's work are summarized in the following list.

- A subclass may not eliminate a state that is in the state machine inherited from its parents.
- A subclass may not eliminate a transition that is in the state machine inherited from its parents.
- In a strict inheritance environment, any new data attributes introduced in a subclass can be accounted for within substates of the states of the parent class.
- The substates of a state form a partition of that state.
- Multiple inheritance results in a set of concurrent states in which there is no relationship between the disjoint sets of states.

4 Coverage strategies

Coverage strategies define adequacy criteria for the testing process. They describe a methodology for selecting test cases to be used in the process; but, more importantly, they define a criteria for when to stop selecting test cases. For example, in the testing of procedural systems, a popular structural coverage strategy is to investigate some path between every definition and its use. This definition-use strategy indicates that we will construct test cases so as to cover one path between every definition and some use of that definition. In this section we will consider the types of errors that can be found using this approach and we will define a set of adequacy criteria that will guide in the selection of test cases.

4.1 Error detection power for classes

There are several types of errors that can occur in the implementation of a state-specified class. In this section we will describe those errors we intend to search for using the strategies described in the next section.

- **Missing/Extra states** - A missing state is one that is described in the specification, but that is not found in the implementation. An extra state is one that exists only in the implementation. After each transition, the object should be in a state described in the specification. If it is not then the current state is an extra state which may mean an error in the transition or it may mean that the specification overlooked a necessary

state. If the test cases that are intended to place an object in a particular state are unable to do so then that state is considered to be missing even if some other test cases might have been able to place the object in that state.

- **Missing/Extra transitions** - The execution of an event results in some transition being traversed, even if the transition returns to the same state. Consider a test case that is intended to take an object in one state and transform it into another state. If after execution of the test case the object is not in that target state, we have evidence of a missing transition or a missing state.

- **Transitions with incorrect input/output** - The input to a transition are the parameters to the event that implement the transition and any globally accessible data. If the incorrect input still results in a transition to the appropriate state, this error will not be detected by functional methods. However, any "missing transition" errors should be examined to see if the event is simply receiving the wrong input to transition to the expected state.

  Incorrect output will be directly observable from the execution of the events. Any incorrect output should be investigated and corrected; however, a functional approach will not systematically detect these types of errors.

- **Corruption** - Corruption is a type of error that results when a piece of code appears to execute correctly, but a side-effect occurs and modifies the state in such a way that subsequent operations will not perform correctly. For example, the enqueue message might correctly place an item at the back of the queue, but "lose" the pointer so that the second attempt at enqueueing will encounter a corrupt reference. These are not specification errors and our technique will not guarantee any systematic detection of these types of errors; however, many will be found in the execution of any test suite.

4.2 A hierarchy of coverage strategies

Four coverage methods are discussed below that provide rules for constructing a test suite. Each test suite provides some degree of error detection and carries some cost based on the number of test cases that must be constructed, executed, and evaluated. Figure 1 illustrates the subsumption relationships among the strategies with those at a higher level subsuming all those below them.

4.2.1 State coverage

A minimal strategy would be to develop sufficient test cases that the object is placed in every state defined in the specification. Assuming that the objects have more than one state, this strategy is sufficient to detect the following types of errors:

- *All missing states.* Each test case is intended to place the object in a particular state. If, after the execution of all test cases, the object has not been placed in a particular state, that state is effectively missing.

- *Possibly some extra states.* These test cases will exercise some of the methods of a class. If, by executing the test cases, the object is placed into an undocumented state, this

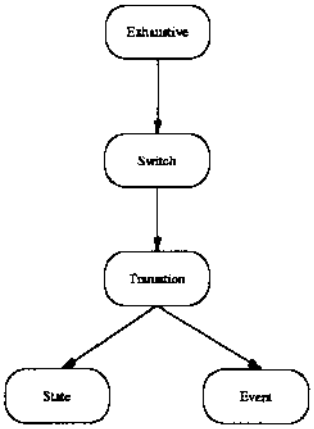


Figure 1: Hierarchy of Adequacy Criteria

extra state will have been found; however, there is no assurance that all or any extra states will be located.

- *Possibly some errors in transitions.* By traversing any of the transitions in the representation some errors such as missing transitions or transitions with incorrect input or output may be found, but there is no systematic coverage of these types of errors.
- *Possibly some instances of corruption.* The fact that some events will be executed in order to place the object in the states implies that some corruption may be found, but there is no systematic coverage.

4.2.2 Event coverage

The event coverage strategy guarantees that every method will be executed at least once. Note that event coverage does not subsume state coverage. It is quite possible to exercise each event without being in certain states at all. The types of errors that we would expect to locate include:

- *Possibly some state errors.* This strategy has no systematic coverage of states therefore any errors found are a matter of chance. Some missing states and some extra states might be identified just in the course of executing each event.
- *Possibly some transition errors.* Each transition is implemented by some event. By executing all events at least once, some transitions are traversed and some errors may be discovered, but there is no systematic coverage of these errors.
- *Possibly some instances of corruption.* By executing methods some instances of corruption may be discovered.

4.2.3 Transition coverage

This level of coverage guarantees that every transition is traversed at least once. This coverage subsumes both state and event coverage. The errors that will be detected include:

- *All missing states.* By traversing every transition, every state is scheduled to be visited. If, after all test cases have been executed, some states are not reached, they are effectively missing.
- *Improved detection of extra states.* Although not every possible parametric value will be used with every transition, by traversing every transition, we would expect to find a higher percentage of states that are not documented than were found using either state or event coverage strategies.
- *All missing transitions.* Test cases are selected and identified for each transition in the representation. If one of the test cases fails, that transition is effectively missing.
- *Some transitions with incorrect input/output.* Although every transition will be traversed, not every possible value for the parameters will be used. It is possible that some incorrect input/output will not be discovered.
- *Improved detection of corruption.* Increasing the range of values over which an event is executed, increases the likelihood that instances of corruption will be identified.

4.2.4 Switch coverage

Switch coverage is a general term for coverage of sequences of transitions. For example, a 2-switch represents a transition into a state followed by a transition out of that state and into a succeeding one. (Chow[1] describes an n-switch coverage where n can be increased up to a "grand tour" case that attempts to traverse the entire representation in one case.") In addition to subsuming the transition coverage strategy, this strategy will improve the possibilities of detecting corruption and extra states. It does this at the cost of many more test cases and represents a small improvement in detection power. However, it does define a systematic approach to gradually improving detection power until resources are exhausted.

4.2.5 Exhaustive coverage

Actually a fifth level of coverage is possible. Exhaustive coverage exercises each event over its entire range of parametric values. This results in all possible paths in the state representation being covered. However, even one cycle in the representation can make this an impossible level to achieve since the number of cases required may become infinite.

5 Testing Methodology

This section brings together the information on object state, types of errors, and coverage strategies to define a methodology for the functional testing of classes and object-oriented applications. The methodology provides a systematic technique for constructing test cases and, perhaps more important, a context for deciding when a sufficient number of cases have been developed.

5.1 Preparing to test

The methodology assumes that a state representation for the class is available. It also assumes that, since we are considering the specification of the class, all of the states are observable. Either there are accessor methods that return state information or the return values on modifier methods provide sufficient information to determine the state of the object. By concentrating on the observable states, we do not need to add instrumentation to the methods as suggested in [6].

The methodology also assumes that a coverage strategy has been selected. This strategy will determine how the test cases are constructed and when a sufficient number have been created. The strategy should at least be the "all transitions" strategy and this is the strategy that we will assume in the remainder of the paper. One of the reasons for developing the hierarchy of coverage strategies is because different levels of testing are consistent with different goals for the class being tested. A class will often migrate from being a prototyping class, with little formal testing, to a production class with a reasonable amount of testing, and finally to a library class requiring intense testing. The hierarchy of strategies provides a clear migration path in which the test suite for a class can evolve in parallel to the evolution of the class.

5.2 Constructing test cases

Each test case is a sequence of messages to an object, but the intent is to test a class. The initial message in each test case constructs an instance of the class and places the object that represents the instance into a specific state. If the "all transitions" strategy has been selected then we can view this "constructor" message as a transition into the state representation for every possible different initial state. Our coverage strategy would imply that we should test each of these transitions or ways of creating instances. It also means that there is no need to create large numbers of different objects with different initial values if all of the newly created objects initially belong to the same state.

The test cases can be constructed by concatenating sufficient messages to reach the initial state in the transition which is to be "covered" by the case. The event that implements the transition is then concatenated as the second component in the triple, and finally the expected termination state of the transition is added as the third component in the triple. The input information associated with each transition can be used to automatically build the test cases according to the following algorithm. Although it is not evident from the algorithm, each transition added to the original sequence would be in the form of a message with required parameters. The TerminationState component would include a message or messages that would be used to determine the actual state of the object after it had received the message sequence contained in the first two components of the triple. (This sequence is characteristic of the state rather than the transition and would be constructed for every state in the representation and then used for testing each transition involving that state.) The other part of the third component is an indication of the state in which the object should be, given the sequence of messages sent from the first two components of the test case. A correct test case is one for which the methods in the third component elicit the same state as is stored in the third component.

```
Algorithm TestSuiteBuilder(StateRep,StateMethods) return TestSuite
    TransitionSuite is instance of TestSuite
    For every Transition in StateRep do
        triple.FirstComponent = SelectInitialTransition;
        While (triple.FirstComponent.TerminationState != Transition.OriginState) do
            triple.FirstComponent = triple.FirstComponent + SelectTransition;
        endwhile
        triple.SecondComponent = Transition.event;
        triple.ThirdComponent = Transition.TerminationState;
        TransitionSuite.Add(triple);
    endfor;
    return TransitionSuite;
end TestSuiteBuilder
```

6 An example

Consider a simple *Queue* class that can be thought of as having two design states, *empty* and *not empty*, as illustrated in Figure 2(A). It is obvious from the transitions labeled **Dequeue** that the *Queue* does not represent a finite automaton; however, the additional information available about an object , e.g. the state identification methods described in the previous section, makes this a most useful representation. In the figures only the essential methods are shown to simplify the diagrams.

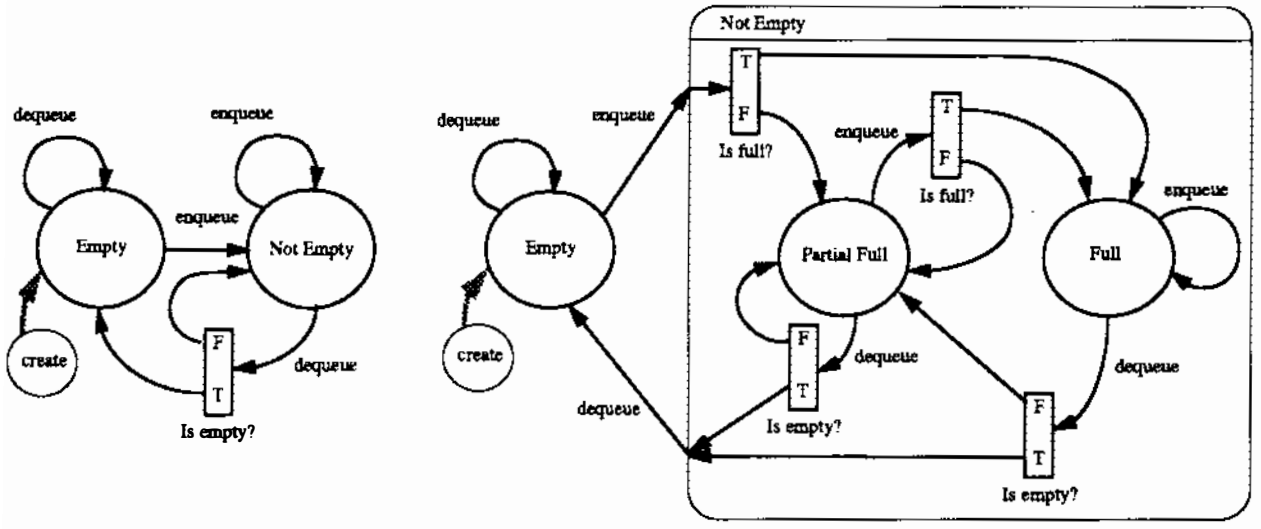
Figure 3 shows three test suites for the *Queue*. The state coverage strategy does not execute the **Dequeue** method at all, while the event coverage strategy does not guarantee that all states of the object are tested nor that a range of values are attempted for the events. The transition coverage strategy does guarantee that all states are entered and that all methods are executed at least once. An **IsEmpty** method would provide sufficient access to the state of the queue.

Consider now a *Bounded Queue* class that is a subclass of *Queue*. The state representation for *Bounded Queue* is given in Figure 2(B). In this example, the substate concept is used to represent the specialization of the *not empty* state into two substates *partially full* and *full*. The test suite for *Queue* can be reused with only a few modifications and the appropriate additions. The modifications specialize the test cases from *empty* to *not empty* to become a test case from *empty* to *partially full* and the test cases from *not empty* to *empty* are also modified. A complete functional test suite is given in Figure 4.

7 Conclusions and future work

The work presented in this paper is one of several avenues of research on testing object-oriented systems being pursued at Clemson University. The goal is to develop the algorithms to automate much of the testing of object-oriented software.

The approach presented in this paper is a state-based testing technique that utilizes a



(A) State diagram for a QUEUE.

(B) State diagram for a BOUNDED QUEUE.

Figure 2:

"All states" test suite for *Queue*
1: <create><dequeue><*empty*>
2: <create><enqueue><*not empty*>

"All events" test suite for *Queue*
1: <create><dequeue><*empty*>
2: <create><enqueue><*not empty*>

"All transitions" test suite for *Queue*
1: <create><dequeue><*empty*>
2: <create><enqueue><*not empty*>
3: <create, enqueue><dequeue><*empty*>
4: <create, enqueue><enqueue><*not empty*>
5: <enqueue, enqueue><dequeue><*not empty*>

Figure 3: Test suites for the *Queue* class

"All states" test suite for *BoundedQueue*
1: <create><dequeue><*empty*>
2: <create><enqueue><*partially full*>
3: <create,enqueue><enqueue><*full*>

"All events" test suite for *BoundedQueue*
1: <create><dequeue><*empty*>
2: <create><enqueue><*partially full*>

"All transitions" test suite for *BoundedQueue*
1: <create><dequeue><*empty*>
2: <create><enqueue><*partially full*>
3: <create, enqueue><dequeue><*empty*>
4: <create, enqueue><enqueue><*full*>
5: <create, enqueue, enqueue><dequeue><*partially full*>
6: <create, enqueue, enqueue><enqueue><*full*>

Figure 4: Test suites for a *Bounded Queue*

state machine representation, which is usually constructed during the design process, to guide the creation of the functional test suite for a class. A hierarchy of adequacy criteria, stated in terms of degree of coverage of the state representation, was presented. The hierarchy describes the relationships among the various testing strategies and provides a natural migration path for the increasingly rigorous testing needed as a class evolves into a more reliable, and more relied upon, component. A testing methodology, that provides a high degree of automation, was also presented.

References

[1] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4, May 1978.

[2] Phyllis G. Frankl and Roong-ko Doong. Testing object-oriented programs with ASTOOT. In *Quality Week 1991*, San Francisco, CA, 1991. Software Research, Inc.

[3] Mary Jean Harrold and John D. McGregor. Hierarchical incremental testing. Technical Report TR91-111, Department of Computer Science, Clemson University, 1991.

[4] John D. McGregor and Douglas M. Dyer. A note on inheritance and state machines. Technical Report TR93-114, Clemson University, 1993.

[5] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[6] C. D. Turner and D. J. Robson. The testing of object-oriented programs. Technical Report TR-13/92, University of Durham, 1992.

True Stories: A Year in the Trenches with MOTHER

Joe Maybee

Tektronix, Inc.
Graphic Printing and Imaging Division
P.O. Box 1000, M.S. 63-424
Wilsonville, Or 97070
maybee@pogo.wv.tek.com

Portland State University
Department of Computer Science
P.O. Box 751
Portland, Or 97207
maybee@cs.pdx.edu

Abstract

Software test fixtures have been used for many years by quality assurance groups, but what advantages do they really provide? This paper examines the use of a test fixture (MOTHER) in a concurrent engineering environment and considers it's impact on an embedded system project.

Keywords and Phrases: real-time embedded systems, automated test methods, requirements testing, electromechanical systems.

Biographical: Joe Maybee has been building real-time embedded systems for Tektronix, Inc. since 1978. Recently, Joe has focused on software quality in reactive systems and the accurate definition of software requirements. Joe also teaches Software Engineering at Portland State University.

Joe is the recipient of an *Outstanding Achievement Award for Innovation* from the *1992 Pacific Northwest Software Quality Conference* for his work with the MOTHER test system, and first-place winner at the *1993 Tektronix Engineering Conference* for his analysis of data gathered about the MOTHER system.

True Stories: A Year in the Trenches with MOTHER

Joe Maybee
Tektronix, Inc. / Portland State University

Introduction

Concurrent engineering is becoming a fact of life for American industry. It therefore becomes necessary to adapt software engineering strategies to the realities of concurrent engineering.

One approach to testing strategies for concurrent engineering projects, MOTHER (Maybee's Own Test Harness for Evolving Requirements), was presented at the *Ninth Annual Pacific Northwest Software Quality Conference.*[1]

The time has come for an accounting as to the success of this approach.

Background

The Quality Imperative

Concurrent engineering provides a formidable set of problems for the engineering environment, but concurrent engineering of Real-time systems is even more daunting. The most complex of these problems are problems dealing with ensuring the quality of a real-time system that is evolving.

Real-time systems are critical systems. They require special attention because of the time domain requirements. The time domain requirements exist in real-time systems because of the consequences of events not being serviced within a specified time. Almost always these the consequences are hazards to either humans or equipment.

Real-time systems absolutely, positively must be right.

High volume systems are difficult to service. The number of units shipped can be, well, astronomical. Since there are such a profuse number of systems shipped, they are difficult

1. Maybee, Joe, "*MOTHER: A Test Harness for a Project with Volatile Requirements*", 1991 Proceedings, Pacific Northwest Software Quality Conference, Portland OR, 1991.

to upgrade in a timely and economical fashion: they may as well be in space. Hence, we feel that firmware for our high-volume systems should be "firmware that can be put in orbit."

There is no doubt that for real-time process control systems, software quality is a "*do or die*" situation.

The Test Harness

MOTHER is a test harness for projects with volatile requirements and volatile requirements are a hallmark of concurrent engineering projects. MOTHER is a test fixture designed to support change. MOTHER is also a low-budget approach.

The fundamental MOTHER approach is to use a finite state machine (FSM) method for software requirements specifications (SRS), then embed the tests directly in the SRS. Embedding the tests in the requirement solves problems of traceability for updating both the requirements and the tests. The tests are written using specialized FORTH operators.

A public domain FORTH interpreter, written in C, was modified to accommodate specialized operators that could provide a stimulus to, or read the state of, the system under test.

The FORTH interpreter was also modified to keep a log of the transactions executed and their success or failure. These logs can then be used to automatically generate test exception reports.

The entire system is bound together with shell scripts and Perl programs. An extensive account of how this entire system was built can be found in the previous paper.

The harness itself worked very well. However, the value of this approach needs to be evaluated on the basis of defect detection.

Project X

Project X is an embedded real-time system project that was subjected to MOTHER testing. Defects were tracked via a defect database, and later subjected to analysis to determine the effectiveness of fixtured system testing with the MOTHER test harness.

In particular, this paper concerns itself with the viability of fixtured testing as a key element of a multiple testing method strategy. As a result, the analysis of the defect database also examines whether defects found by the test fixture could have been found using alternate methods, and at what expense.

Project X Metrics

In evaluating the performance of the fixture, it is necessary to understand the scope and magnitude of both Project X and the fixture.

The raw facts:

The test fixture code consists of:

- 9,564 lines of C/C++ code. [1]

The product code consists of:

- 28,936 lines of code total.
- 22,951 lines of high level language code.
- 5,985 lines of assembly.

The Specification consists of:

- 1,487 requirements.
- 44,814 lines.
- 116,024 words.
- 1,277,568 characters.

The Tests consist of:

- 1,572 tests.

Derived numbers

Now that we have the raw numbers, let's look at some interesting derived numbers:

- Roughly 20% of the code for Project X was assembler. That's one in five lines.
- There were approximately 30 lines per requirement. (The requirement includes the test description.)
- In terms of words, there were 78 words per requirement, including the test description.
- There was one line of test harness code for every 3 lines of product code.
- Production yielded about 20 lines of product code per requirement.

1. Code counts do not include blank lines and comments.

What the COCOMO model would predict

If we were to plug some of these numbers into a software model, we would get some interesting results that we could compare with the actual numbers.

First, let's look at what Boehm's COCOMO model[1] would predict for the *test fixture*:

```
Model mode: organic
Model size: intermediate (9564 lines of code)

Total effort:   25.7 man-months       [152 man-hours/man-month]
Total schedule:  8.6 months           [standard calendar months]

Distributions:                 Effort          Schedule        Personnel
                             (man-months)      (months)       (on-board)

     Plans and requirements: (06%)  1.5     (11%)  0.9          1.6
   Product design:           (16%)  4.1     (19%)  1.6          2.5
   Programming:              (65%)  16.7    (59%)  5.1          3.3
     Detailed design:        (25%)  6.4
     Code and unit test:     (40%)  10.3
   Integration and test:     (19%)  4.9     (22%)  1.9          2.6

Programmer productivity during code and unit test phase:  930 DSI/month.
[DSI = Delivered Source Instructions]
```

As a matter of record, it took an estimated 10 man months to accomplish this. Now let's look at what the same model would predict for the *product code*:

```
Model mode: embedded
Model size: medium (28936 lines of code)

Total effort:   204.2 man-months      [152 man-hours/man-month]
Total schedule: 13.7 months           [standard calendar months]

Distributions:                 Effort          Schedule        Personnel
                             (man-months)      (months)       (on-board)

     Plans and requirements: (08%)  16.3    (32%)  4.4          3.7
   Product design:           (18%)  36.8    (34%)  4.7          7.9
   Programming:              (54%)  110.3   (40%)  5.5          20.1
     Detailed design:        (26%)  53.1
     Code and unit test:     (28%)  57.2
   Integration and test:     (28%)  57.2    (26%)  3.6          16.0

Programmer productivity during code and unit test phase:  506 DSI/month.
[DSI = Delivered Source Instructions]
```

1. This is from a home-brew program that I wrote that implements Boehm's COCOMO model. For more information on what these fields really mean, see Barry Boehm's epic: *Software Engineering Economics.* Indented sections of the COCOMO model output are subsections of adjacent project phases.

Again, for the record, it took an estimated 72 man-months (six man-years) to accomplish this. So far, so good. These appear to be phenomenally productive engineers.

The reasons for this phenomenal productivity are twofold:

1. The technical problems for the software engineers was not extreme at all. In the case of Project X, all software problems were classic problems with textbook solutions. There were no ugly surprises lurking in the wings. All software was constructed using known methods and techniques.
2. The use of templates allowed us to generate lots of code very quickly (at least on the part of test fixture development.) In the test fixture, the installation of a new operator was almost trivial: the engineer could copy an existing operator (or use the standard template) and generate a new custom operator very quickly.

Test Strategy for Project X

The firmware was evaluated using a threefold approach:

1. The firmware was subjected to exhaustive functional testing using the MOTHER test fixture.
2. The Project X firmware was then subjected to statistical testing in the QA laboratory. (Alpha testing.)
3. The Project X firmware was then released to Beta sites for testing.
4. Finally, Project X was released to the customer. Defects discovered in the field were logged for future releases.

Figure 1 illustrates the test sequence for Project X.

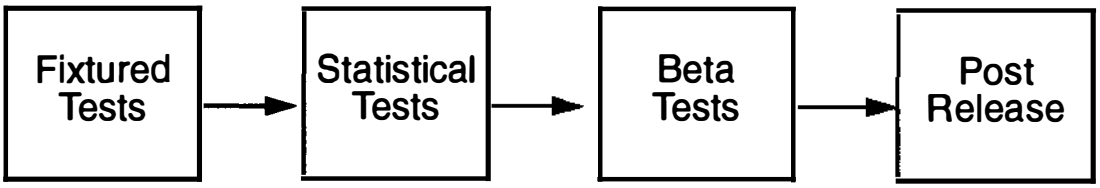


Figure 1: Test sequence for Project X software

Now let's look at approximately how many defects were discovered using the various methods:

- 9% of the defects were found by the software engineers themselves.
- 64% of the defects were discovered by the test harness.
- 12% of the defects were discovered by the statistical testing lab.
- 14% of the defects were discovered by beta testing.

64% of all defects were discovered by the test harness itself. This also means that 64% of all defects were detected before the firmware went into statistical testing or beta testing. We were able to provide both the statistical testing lab and beta sites with a list of known bugs. Our beta sites only detected 14% of the bugs.

Coverage

Let's begin by considering the types of tests that existed for the requirements. Table 1 shows the number of tests that have fully automated, partially automated, and un-automated tests (tests that have no automatic tests or are only partially written).

TABLE 1. Breakdown of tests for Project X

Test Type	Number of Requirements	Percentage of Total Requirements
Automatic	801	53.8%
Manual assistance required	121	8.13%
No automatic test	314	21.1%
Tests partially written	251	16.8%

There were a total of 1487 requirements and 1572 tests.

In looking at the number of tests and the number of requirements, we notice an obvious discrepancy: they don't match. There are three primary reasons for this:

1. A requirement may have multiple tests. For instance, if the specification indicates that a set of disjoint ranges are invalid, each of those ranges is checked. This causes multiple tests for a single specified requirement.
2. A requirement may have a test only partially automated. These tests require manual intervention, and are partially automated.
3. A requirement may not have an automated test written at all.

So 61.93% of all requirements had either automatic or manually assisted tests. The others (37.9%) had tests that were not automated. The primary reason for the number of unautomated tests ("no automatic tests" and "tests partially written") was that it was difficult or impossible to interface the test fixture with the hardware necessary to perform these tests.

A major contributing factor to the number of unautomated tests was that there was only a single test fixture. The test fixture and test operators were kept busy running suites or debugging automated tests: this made it difficult or impossible to find time to develop the new operators and hardware interfaces necessary to implement the final 38% of the tests.

There were two primary lessons learned from this:

- To develop a test harness, there needs to be a harness to develop on. Make sure a development harness is available at all times.
- To develop a test harness, design the product to accommodate an automated test strategy. It should be a fundamental requirement of the product.

Characteristics of Defects Found Using Fixtured Testing

In general, fixtured testing would catch defects that:

- Are detectable at the physical level. That is, defects that are detectable by casual operators.
- Occur systematically. Systematic defects are defects that occur in every instance of the test, and are not intermittent.
- Occur at boundary values.
- Occur as sequence errors. Sequence errors are defects that occur as the result of a combination of operations, rather than a single operation.
- Are foreign language defects. The test fixture checks foreign language messages against an internal list of correct messages.

In general, fixtured testing would not catch defects that:

- Occur intermittently. If the error manifested itself while the fixture was running, the error would be caught. However, in general, the fixture does not trap this class of defect.
- Occur only in special environments (certain exotic hosts, etc.) The test fixture ran in a limited environment.

Characteristics of Defects Found Using Statistical Testing

Usually, statistical testing would catch defects that:

- Are detectable by casual operators: that is, operators who interact with the system on a physical basis.
- Manifest themselves in an intermittent fashion.

Statistical testing would not ordinarily catch defects that:

- Manifest themselves because of interaction with the system environment. (This would change at a later date, as statistical testing moved into the system integration phase. For the time covered by this paper, it is fair to say that this class of testing was negligible.)
- Dealt with boundary value conditions. For the most part, statistical testers do not run machines on boundary values. Statistical testers most often deal with the ordinary values.
- Sequence errors: Statistical testers do not aggravate sequence errors beyond ordinary operation. Although statistical testing can detect some sequence errors, this tends to fall outside the scope of statistical testing.
- Foreign Language defects. Statistical test operators do not (in general) run tests in foreign language mode.

Characteristics of Defects Found Using Beta Testing

Beta testing would catch defects that:

- Occur in special environments. Beta testing allows the "customer" to apply the system to their own environment (exotic hosts, etc.)
- Occur systematically, as opposed to intermittently. Intermittent defects are sometimes "written off" by beta testers as unexpected but not incorrect behavior.
- Occur in the normal, not exceptional, operation of the machine. Beta sites concern themselves primarily with causing the machine to operate in the correct fashion, rather than trying to initiate exceptional conditions.
- Foreign Languages. This presumes that the beta sites employ native speakers.

Beta testing would not catch defects that:

- Dealt with boundary value conditions. "Customers" are more concerned with normal operation that exceptions.
- Sequence errors. Most Beta sites adhere to standard sequences for interacting with the system. Usually this is because the Beta sites are more interested in integrating the system with their own: their prime objective is to make the system work, not exercise the system.

The Economics of Defect Detection Rates

The MOTHER test fixture proved itself to be an effective, economical addition to the overall verification strategy. In an analysis of the defect detection rates for Project X, we determined that of the 64% of defects discovered by the MOTHER test fixture, 49% of the defects found by MOTHER probably would have been found at later stages of testing (statistical or Beta testing.)

Specifically, of the 64% of the defects found by MOTHER:

- 27% of those defects probably would have been found in statistical testing.
- 23% of those defects probably would have been found in Beta testing.
- 14% may have gone undetected until after release.

Of the various methods, we determined that the defect detection rates were[1]:

- 0.54 defects/day using our Statistical testing method.
- 0.92 defects/day using MOTHER.

In conclusion, it appears (from Project X data) that fixtured testing can have a nearly twofold defect detection rate for a particular class of error. In this case, the 63% coverage yielded approximately 63% of the total defects detected.[2]

It should be apparent from the characteristics of defects found by each testing method that there is no substitute for statistical testing techniques. Fixtured testing can yield its greatest benefit at the project endgame, where time is of the essence: it can find certain types of defects faster than other methods.[3]

What Worked Well

The MOTHER test fixture performed satisfactorily in several areas, primarily consistency and speed.

The MOTHER system allowed extremely consistent testing. The system was capable of running tests over and over with no variance. This attribute was very useful: all exceptions were re-run to verify the defect before the defect report was submitted to design engineering. The end result was a defect report which, in the majority of the cases, was repeatable at will. In the instance where the defect was not repeatable, the defect was logged as an intermittent defect, and was watched for in statistical testing.

The MOTHER system also allowed fast testing turnaround. The MOTHER test fixture was capable of meeting its time requirements: no more than 48 hours for complete turnaround of all tests, including the tests that required manual assistance.

---

1. The defect detection rate was calculated using the number of defects detected by a method divided by the total number of days for the evaluation period. This approach errs on the side of statistical testing, simply because fixtured testing began at a later point in the project.

2. This direct correlation begs a question: "If Project X had 100% coverage, would the fixture had found 100% of the defects?" The answer is: no. This correlation between the percentage of test coverage and percentage of total defects found is a mystery to the author.

3. We are dealing with a sample size of one project in this instance. We will be watching as additional projects come on-line for continued support for this position.

What Needs to be Improved

The MOTHER system was not without its problems. In particular, control of the time domain, the test language, operator interface and tight coupling to the system under test were problems for the fixture.

Time domain (real-time) problems were aggravated by the test language. Concurrent events had to be specified as sequences organized in time, rather than events with time constraints. The net result was that the test had to be written in a sequential fashion. If the timing relationships caused sequence changes in the system, the tests had to be reorganized to conform to the new sequence. In one instance, this caused a undesirable delay. It would be better to use a descriptive notation for events, then let the test harness compute the sequence of event occurrence.

The operator interface left the operator wondering about the status of the tests. There was no indication of the amount of time elapsed since the last test, no indication of the estimated amount of time left in the suite, and no indication of the number of tests which had failed. It was therefore difficult for the test operator to determine if the system under test was behaving in a correct fashion, and it was equally difficult to determine if an inordinately large number of tests had failed (an indicator of some systematic problem.)

The MOTHER system was tightly coupled to the system under test. MOTHER required direct access to the hardware, and required special operators to be designed for each system to be tested. This made the job of porting MOTHER from project to project more difficult than desirable.

Coming Soon! "Mother2: Son of MOTHER"

The first incarnation of MOTHER has demonstrated that rigorous specification and rigorous testing can provide high quality software. This has been known for some time. What is different about this strategy is that we can accommodate sudden and unplanned changes on the testing side of the process as a matter of course.

Our Quality Assurance group has come to the conclusion that MOTHER was an excellent prototype. However, we have learned some lessons from this prototype, and have decided that we need to "Pitch it and do it over." This stems from a basic philosophy of what we consider to be a successful software engineering strategy: "Adapt, adopt, or reject."

We have many avenues open to us, so we have decided to:

- Adapt the elements that have value but need to be modified.
- Adopt the successful elements of the previous test harness.
- Reject the unsuccessful elements of the previous harness.

The successor system has been dubbed "Mother2", but in actuality bears little or no resemblance to the previous incarnation.

Some of the features of the new system are:

- Concurrent events capacity. The prior harness had a sequential approach to tracking events. This became problematic when timing changes caused the sequence of events to change. The new test language uses a sequence independent approach to testing.
- New test language. Original Mother2 tests were written in FORTH using extended FORTH operators. FORTH notation is not very intuitive for those of us used to the more traditional languages. The new test language for Mother2 is an assertive language: it is used to specify what a properly functioning system *shall* do. Failure to meet these assertions is detected by Mother2.
- New operator interface. Some of the information we displayed on the console was useless. Some of it needs to be presented in a different fashion. Currently these items include: number of tests in the suite currently running, estimated time to completion, number of failures and successes detected, and an interactive interface for tests requiring manual intervention (such as loading special paper, etc.)
- Decoupling between fixture and system under test. Using external configuration descriptions ought to give us the ability to define the configuration of the system under test. This should expand the system usefulness.
- Automated test generation from specifications. The new system uses an automated specification checking and test generation system written in Prolog. This software is experimental, but can generate automatic tests for the new Mother2 test fixture, and check the specifications for certain types of errors common to finite state machine (FSM) specifications (completeness, consistency, etc.)

Summary

Test harnesses can be valuable tools. The payoff is potentially large: even if it is difficult to define or implement everything needed, this example indicates that it is possible to detect a large portion of the defects early in the testing process.

Test harnesses also give quick feedback on functional tests. This quick feedback can be useful in deciding whether or not to release a version for statistical testing. Also, it allows quick compilation of bug lists for release to the statistical testers and beta test sites.

Our analysis of the performance of the MOTHER test harness has reaffirmed our belief that fixtured testing can play a major role in the concurrent engineering environment.

Acknowledgments

There were a number of people who contributed to the success of this project, and have my heartfelt thanks:

- Raul Krivoy, my manager. Raul asked for an analysis of MOTHER data that ultimately lead to this paper.
- Jan Maybee, whose keen eye for grammatical and spelling mistakes saved me much time.
- The various members of my QA group who gave me constant encouragement.

Without their assistance and help, you wouldn't be reading this paper.

TEST CONTROL: An Alternative to SCCS/RCS for Computer Aided Software Testing (CAST)

by

Owen Richard Fonorow
NCR Corporation
1100 E. Warrenville Road
Naperville, Illinois 60566
uunet!att!iwtqg!orf

Abstract

The BUSTER™ Test Manager introduces *Test Control*™, a novel mechanism for administering software source trees with large numbers of directories and files, including binary files. Previously, the lack of a suitable administration mechanism left many CAST source trees without any formal control.

Test Control uses a relational database and treats multiple files and directories in tests as one object. This abstraction makes it more efficient to use than the usual file-based source control mechanisms (e.g. SCCS) which manage files in tests individually. *Test Control* features a standard test format, a generic test executor, and it provides a Test Database where all test results are stored for audit purposes. BUSTER also includes utilities for managing the end-to-end testing process.

About the Author

Owen Richard (Rick) Fonorow graduated from the U. S. Air Force Academy in 1976 and obtained his M.S. in Computer Science from the University of Arizona in 1984. He joined AT&T Bell Laboratories as a Member of the Technical Staff and moved into NCR during the recent merger. Rick currently works as a developer in the software tools group that develops and markets both the QUARTZ Remote Terminal Emulator (RTE) software for performance analysis and the BUSTER Test Manager. Rick was a member of the team that invented BUSTER and holds a BUSTER patent.

TEST CONTROL: An Alternative to SCCS/RCS for Computer Aided Software Testing (CAST)

by

Owen Richard Fonorow
AT&T/NCR Performance Analysis and Tools

1. INTRODUCTION

Those responsible for software quality know how difficult it can be to justify the cost of testing. It is impossible to prove, apriori, the amount of testing a software product requires. How can the outcome of any activity that seeks to prevent an unknown number of unlikely events be quantified or predicted? How much spending for national defense prevents war? Will hiring more policeman prevent crime? How effective is the security video camera? Or, how much should a particular software product be tested?

There are no pat answers to these kinds of questions. The problem is one of unprovable (hidden) benefit versus visible cost. The answer is usually determined by fear, i.e. the risk of being wrong. Ironically, the better the testing process has been, the more invisible the benefits of testing become over time.

Activities that are perceived as costly without tangible benefit are not likely to be approved, especially in the absence of specific product quality requirements. Advocating a large testing budget may seem to suggest that expectations are low. In the case of software development and testing, there is a certain logic in releasing a lower cost (i.e. untested) version of a software product, and then letting customer "feedback" dictate future corrective effort. This is obviously expedient. Because of these forces, testing budgets are more likely to shrink than expand.

This paper seeks to explain why the usual software administration mechanisms have not been successful administering CAST source code. Both the Source Code Control System (SCCS) or the Revision Control System (RCS) are arguably quite good for their purpose, i.e. controlling

product source code. But, for perhaps unobvious reasons, these tools become quite expensive in terms of administrative overhead when used to control the kind of software and source code changes that are common in CAST suites. Thus, the formal control of CAST suites is rare because software projects are generally unwilling to tolerate high overhead costs related to the the development and maintenance of their automated testing source code.

1.1 Differences between CAST and Product Source

There are subtle differences between the product and CAST software that can dramatically impact the cost of formal control. These differences may be characterized as follows:

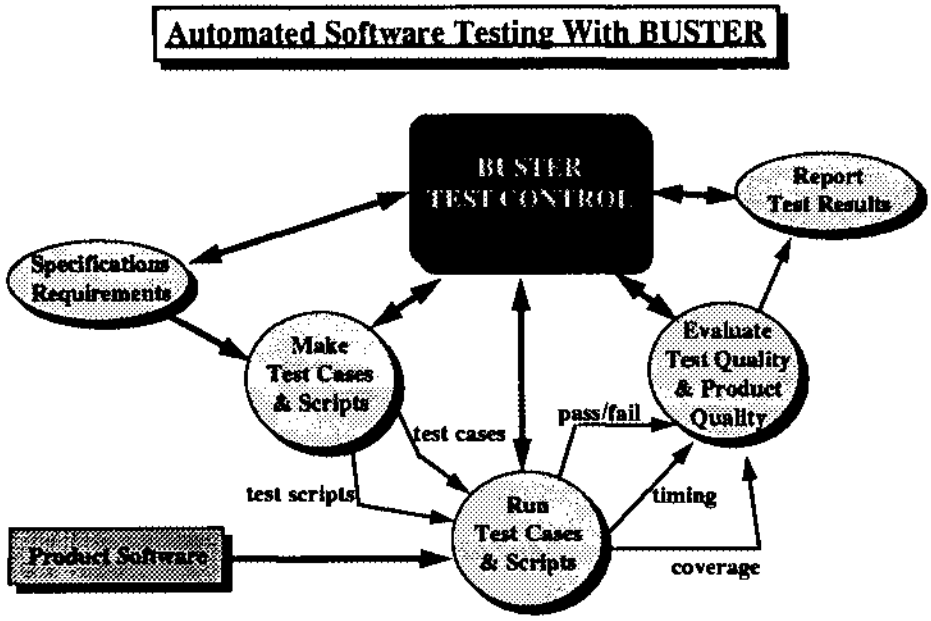
- The CAST source code is usually much larger and may have an order of magnitude more files and directories,
- The CAST software is more reactive; i.e. small product changes are magnified and may have a large impact on the tests,
- Testing environments do not always have direct access to a configuration management database.

When these differences apply, it can be shown that the cost (in terms of overhead) associated with formal control of CAST source is likely to be orders of magnitude larger than the cost of controlling the source for the product being tested; if the same file-based configuration management mechanisms are used in both cases.

2. BUSTER OVERVIEW

There is a less costly method of formal control specifically designed for the needs of CAST software. The BUSTER™ Test Manager is a collection of UNIX system software testing tools based around a low cost formal control mechanism. The tools are designed around a common test script format and there are utilities for running tests and storing results. A major feature of the software is the ability to create a catalog of tests and test results using a commercial relational database.

The unique BUSTER administration paradigm, here called *Test Control,* provides a low-overhead control mechanism for the storage and management of vast numbers of tests. An important distinction of Test Control is that unlike *file-based* configuration management and control systems, the BUSTER administration mechanism treats multiple files and directories as one object — *the TEST.*



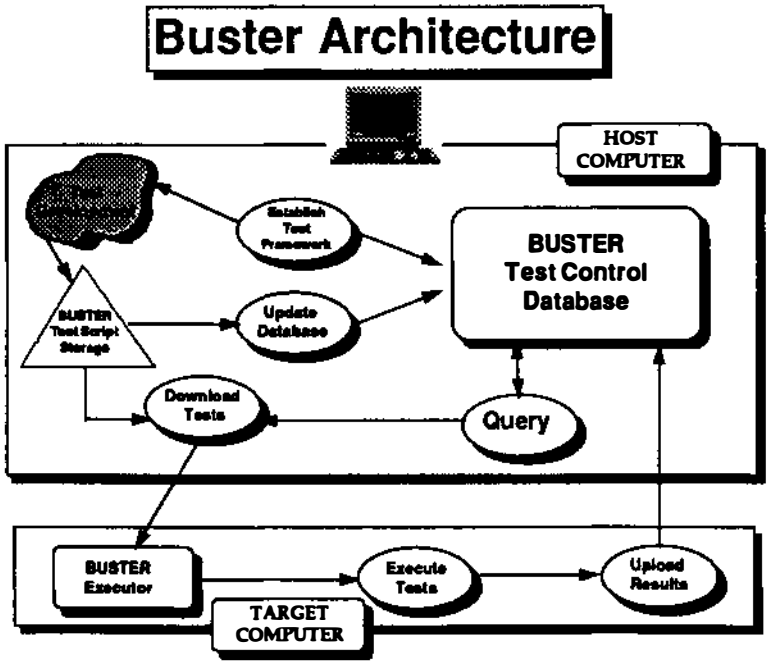
BUSTER was designed by a team of system testers faced with the prospect of testing the UNIX operating system across a multitude of configurations. The idea behind Test Control resulted from the recognition that existing file-based software control mechanisms were not a good fit for test automation software because there are potentially so many files and directories in automated testing suites. The *TEST* entity is an abstraction that simplifies administration and control.

2.1 Host versus Target Machine

The BUSTER software is logically divided into two parts; the *HOST* and *TARGET* executables. The BUSTER *HOST* software features a relational test database that provides knowledge about the entire test suite. The database promotes easy retrieval. The database is automatically updated so testers are generally not concerned with it.

There is a centralized location for official tests and local developer nodes for test development. There is a permission scheme so that developers can to use their private nodes to update or replace official tests. This capability, and many others, can be administratively controlled through the BUSTER "*is allowed*" test control feature.

The TARGET software runs tests. More formally, the software controls test execution. The test may execute based upon action taken by the UNIX shell, or some other UNIX playback tool, or the executor may run a script that prompts a tester who is executing the test manually. Test results are stored and reported similarly in all cases. Testing is normally unattended (automatic), and tests are often run across multiple configurations simultaneously.



The BUSTER test executor expects a test script file to be in a standard format. The test script contains documentary and executable sections. Test developers do not have to learn a specialized testing language because BUSTER accepts and runs plain UNIX system shell programs. Execution times are associated with tests so that those that take too long are aborted and the next test is started. All input/output is logged separately from the test results. Analysis time is minimized because the output spool files do not always have to examined. Succinct test results are transported back to the *HOST* and are recorded in the database. This provides a complete verification history for auditing purposes.

The popularity of BUSTER with Test Control is due to the flexibility of the software, the way it reduces the monotony and effort involved with

testing, and because it promotes test sharing and reuse. An important associated feature of BUSTER's *Test Control* is that it provides a completely flexible *Test Method*. There are few restrictions as to how tests are designed or structured, or what they do.

The BUSTER Test Manager also includes UNIX system software utilities for controlling and managing the testing process. Testers can use the many facets of BUSTER to help administer, develop, debug, run, organize, change, document, gather and otherwise manage large numbers of tests. The BUSTER software also tracks the progress of testing by generating regular reports.

3. ON THE NATURAL SELECTION OF SOFTWARE MANAGERS

Dr. Boris Beizer, an authority on software testing and test automation, puts forth the proposition that test automation is becoming mandatory because markets increasingly demand perfect software products and because manual testing methods are likely to leave faults undetected. Beizer states that automated test software *should be controlled as stringently as the product software under development*.[1]

Dr. Beizer also notes that today this imperative escapes the majority of software development managers. He estimates that "75% of the managers responsible for software development do not recognize the need to invest in testing," much less the need for test automation. Beizer predicts that poor software quality will cause products to fail because competitor's products developed using test automation will have far fewer bugs. This process, much like natural selection, will in turn cause development managers that do not favor testing to lose their jobs — probably without realizing the hidden reasons why. Development managers that survive this slow selection process will be those that invest in testing and test automation.

We accept Dr. Beizer's test automation imperative, but argue that comprehensive configuration management for test suites, although a laudable goal, can be very expensive using file-based tools such as SCCS or RCS. Ordinary SCCS software control systems generally assume stable product source directory structures. These tools were not designed for use with hundreds or thousands of test directories, each of which may contain scores of additional files (including binary files) and directories. Most do not handle the arbitrary deletion of directories well. For this reason they tend to break down when faced with typical automated testing software where tests routinely contain multiple directories.

Those who forge ahead with file-based solutions soon discover that a relatively minor change to the software under test can have a surprising impact on the test suite. In our environment, we find that test failures are often due to a problem with the test itself, and do not indicate a problem with the product under test. In our experience, the percentage of problems that are test-only problems (and do not require product source changes) is more than 80%. In other words, more than 80% of the test failures seen are the result of problems with the tests. Therefore, most of the source code changes that are made in reaction to test "failures" are made to one (or more) files in the test. As will be shown, ordinary source control mechanisms soon break down; becoming the testing bottleneck. As evidence for this contention, there is the scarcity of existing UNIX system Test Manager software packages on the market based upon ordinary SCCS/RCS version control. Although some attempts at Test Managers based on SCCS have been made, there are no successful ones to our knowledge.

4. BUSTER TEST CONTROL EXPLORED

Test Control is deliberately a low-cost administrative mechanism designed to meet the needs of large CAST suites that contain thousands of files. Test storage and test update are managed, but Test Control has no mechanisms for keeping track of versions of tests. Nor does it provide a comprehensive database of the changes made to tests.

However, Test Control does have features that are not usually found in the more common software control systems. For example, unlike ordinary SCCS, the Test Control mechanisms support arbitrary directory deletion, storage of binary files, and there is a standard interface to a relational test database. The database contains information about every test (including purpose, owner, last modification date, last modified by, etc.), and the database is automatically maintained using the test as the source of information.

The basic elements of Test Control are:

- Tests are an abstraction that contain multiple files and directories,
- Tests are self-documenting and based around a standard test script format,
- There is one official version of every test, and
- All tests are stand-alone objects.

Although Test Control provides a clear alternative to the more common forms of source control, we think it offers the same fundamental benefits of formal control. Test storage, for example, is centralized making tests easy to find. Developers have their own private nodes, and the update of official tests can be controlled through a special permission mechanism. And there are standard reports.

4.1 The TEST Directory

BUSTER Test Control abstracts separate files and directories into the notion of the TEST. That is, BUSTER treats collections of files (of any kind) as a logical unit and this abstraction provides many benefits. Overhead is greatly reduced because there is no need to control all the files individually.

The abstract notion of the TEST turns out to be useful outside of Test administration and management. As mentioned, the notion supports the complete flexibility of testing method so the BUSTER *Test Executor* is not restricted to testing a narrow class of technologies. The BUSTER Executor can run tests written in virtually any language, or it can be used to run with any existing capture/playback test tools. The ability to handle directories is important because there are many existing UNIX system capture/playback tools that use directories. One such tool is XRunner® from Mercury Interactive. BUSTER with Test Control makes it easy to store and retrieve (as well as control and run) official copies of multi-leveled directory tests.

4.2 BUSTER Test Case Format

Test Control is based around a standard test case format that is patterned after the IEEE 829 Testing Standards.[2] The standard format promotes a testing framework that binds the various facets of BUSTER together. A single file in the test directory, called the *test script*, is required to be in this format. The self-documenting format reduces the need for other paper documentation. The test information database also has standardized fields, and these standards make it easy to update the database automatically without test developer involvement.

The standard format consists of sections like the following:

ID	The unique testid
CONTACT	Owner of the test
PURPOSE	One line purpose of the test

METHOD	The complete high-level description of the test
KEYWORDS	Important words for database queries and retrieval
REQT	One or more product requirements this test verifies
DOC	Any associated documentation outside the test
SHELL	The command interpreter or language to execute
HCONFIG	Any unique hardware configuration requirements
SCONFIG	Any unique software configuration requirements
LIBRARY	Shared files in the test library required to run the test
SETUP	Executable code to set up the test
PROCEDURE	Executable code that runs the test cases
CLEANUP	Executable code that cleans up after the test completes

Figure 1. Sample BUSTER Test Script Sections

There are several other standard fields, but only ID and PROCEDURE are absolutely required.

4.3 Official Tests

The idea of the official test is arguably the most important feature of Test Control. There is a single "official" version of the test, and it is stored in a centralized, public location. The BUSTER software can detect changes to tests, i.e. tests that differ from the official copy. There is one official version, and no other "versions" of the official copy are maintained (other than through normal backup procedures). If the test result is marked official then project management knows that the test result is from running the version of the test in storage; not some private, "enhanced" version.

Generally, tests are not run directly from official storage. BUSTER provides tools for "downloading" tests (i.e. copying from official storage and bundling then into an archive) and "uploading" changed tests from the *TARGET* back to the *HOST* database machine.

The notion of an official test encourages test generalization. The same test procedure, for example, could be used to verify an application running on different databases. (As opposed to maintaining different versions of the same test procedure for different databases.) There is a test *setup*. The *setup* could be coded to use an environment variable to signify the type of database to test. The *setup* would then create the correct database, e.g. either an ORACLE™, INFORMIX™, INGRES™, or SYBASE™ etc., prior to testing. The test could then run the same test procedure in all cases, perhaps differing only in minor details, based on the environment.

The test *setup* is also used to run the same "version" of the test across different lab configurations. The *setup* can be used to account for various networking idiosyncrasies and other variable factors that do not really affect the test procedure. There is also a *cleanup* that should restore the target environment to a quiescent state.

Occasionally, different "versions" of the same test are required. In these cases, a completely new test is created. The old test may be copied, adjusted, and given a new test identifier. Less frequently a "library" of shared routines is created.

4.4 BUSTER Tests are Stand-alone Objects

Another central (and perhaps counter-intuitive) theme of Test Control is the notion that every test should be able to stand alone and run by itself. A test should not depend on other tests being run, or on the order that other tests are run. Such orderings and dependencies can be established within a test, and using BUSTER terminology they are called "test cases". There is no limit to the number of test cases in a BUSTER test.

Sharing to save space can create other problems for CAST testing software. Shared library routines are supported, but their use should be minimized. To see the reason for this, one needs to understand the drawbacks in the more obvious "hierarchical" approach. Tests that are "built" on hierarchies and share large amounts of code become dependent on the hierarchy.

In practice, tests will sometimes undergo change to reflect changes in the product. It is as naive to think tests never change as to think requirements never change. In the case where much or all of the test suite is dependent on hierarchical libraries, a modest source change can force a library change. It is easily shown that shared library changes can have a cascading affect. These changes not only affect the few tests that need to be updated, they in essence affect *every* dependent test, tests that otherwise would not change. Today, the cost of personnel trying to maintain test suites with sweeping dependencies far outweighs the cost of disk space.

5. NATURE OF THE BEAST

The differences between the CAST testing software and the associated product software are not always obvious. The product source code usually remains on the load building machine with the configuration database. The product is first built. That is, the product software under

test is usually transformed into executable files. These "unchangeable" executables (or binaries) are transported to the testing environment. On the other hand, the usually large CAST testing suites are normally transported in their entirety to target machines as source code.

The CAST software seems to change more often as the result of test failures. It is expedient to update tests on target machines because problems are easily fixed "on the spot". But in testing environments, the changes made to one or more test files are often made without the benefit of the source control database. Contrast this with the product software that arrives on the target machine with little expectation of change.

The problem becomes even more difficult because there can easily be an order of magnitude more files (and directories) in the test software than the product source tree. Looking at this from another angle, "official" changes to the product are normally made to the fewer source files on machines where the SCCS/RCS database can be accessed via normal change control mechanisms.

CAST software is reactive to changes in the product under test. Placing the large CAST source under strict change control is of marginal value. If the product change control mechanism "fails" and the product is then allowed to behave differently, many tests may need to be updated to reflect the new reality. At this point, there is no benefit from making it more difficult to change the CAST software; it only increases the cost.

As an example, lets say that your project *does* want to institute regular change and version control over the CAST testing software. If the configuration management database is not directly available on target machines, it becomes very difficult to keep track of changes to individual test files after changes are made. Without some mechanism equivalent to the TEST abstraction, some other way to "remember" which files in tests need to be checked-out, updated, and checked-in is required.

The following example is typical. The tester prepares for a test session by downloading a suite containing 250 tests to the target machine. (*These 250 tests are abstractions that may in fact contain more than 1000 source files and directories.*) The tester commonly loads this suite on more than one target machine. If only 15 of these tests have to be changed, perhaps in response to changes in the new load of the software under test, there may be several dozen individual files involved. The trick, without BUSTER Test Control, is keeping track of the individual file changes.

The product under test is always changing and the potential always exists for a large scale change. A large scale impact might affect 50-200 of the 250 tests in this example. Without going into detail, suffice it to say that large scale CAST changes can happen if all tests must go through the same user interface, e.g. if ordinary capture/playback tools are used, and if all tests pass through a startup screen that either changes or goes away.

The BUSTER "official test" control mechanism, on the other hand, detects changes on target machines, "marks" all the changed tests, and can also automatically upload the 15 tests for transport back to official storage on the *HOST* machine. And because the tester knows which tests have and have not changed from the official version, there is no need to transport the other 235 tests (and countless files) back to the host machine for comparison. Similarly, the tester is not much concerned from a control standpoint with individual files that may have changed inside the 15 "corrected" tests.

The point is that when you do not have direct access to the SCCS/RCS database, even relatively small scale CAST changes can become costly. So many files may change, or have to be compared back on the host machine, that it becomes next to impossible to track without (something like) the BUSTER-style "official" test mechanism. The problem is magnified when the same test changes for different reasons across multiple target configurations.

5.1 Test Control and SCCS/RCS Together?

BUSTER Test Control and SCCS/RCS control are not mutually exclusive. It is quite common to put the CAST suite under change/version control at every product release point. This way the CAST versions correspond to major product releases. (Minimally, projects should back up the image of their official test suite for every major release of the product that they plan to support.)

Although it is convenient to simplify tests by breaking them into individual files and storing these files in directories, it is possible, at least in theory, to implement every test as a single file. This should significantly alleviate the administrative control problem with current file-based tools. However, this potential solution has drawbacks.

The "single file" style test would likely have the following attributes: It would tend to be large. It would have to solve the problem of comparing output with a reference file, and it might

contain unprintable (binary) characters. Generally, it would be difficult to understand, analyze, and maintain. Furthermore, there are often many more tests than product source files. Relatively minor product software changes may "break" all or most of the tests. When this happens, these tests have to be updated. Rather than changing one or more modular files in a directory, it becomes necessary to change the single large (and hard to understand) file. There is also the question of where to place test output and results otherwise. A well defined test directory makes it easier to find test output and pass/fail results for analysis.

Although not common, it is possible for a mature project with the luxury of plentiful resources (and understanding project management) to use Test Control in conjunction with ordinary software configuration management tools on a daily basis. This combination would provide on-going version and change control of their CAST software during development.

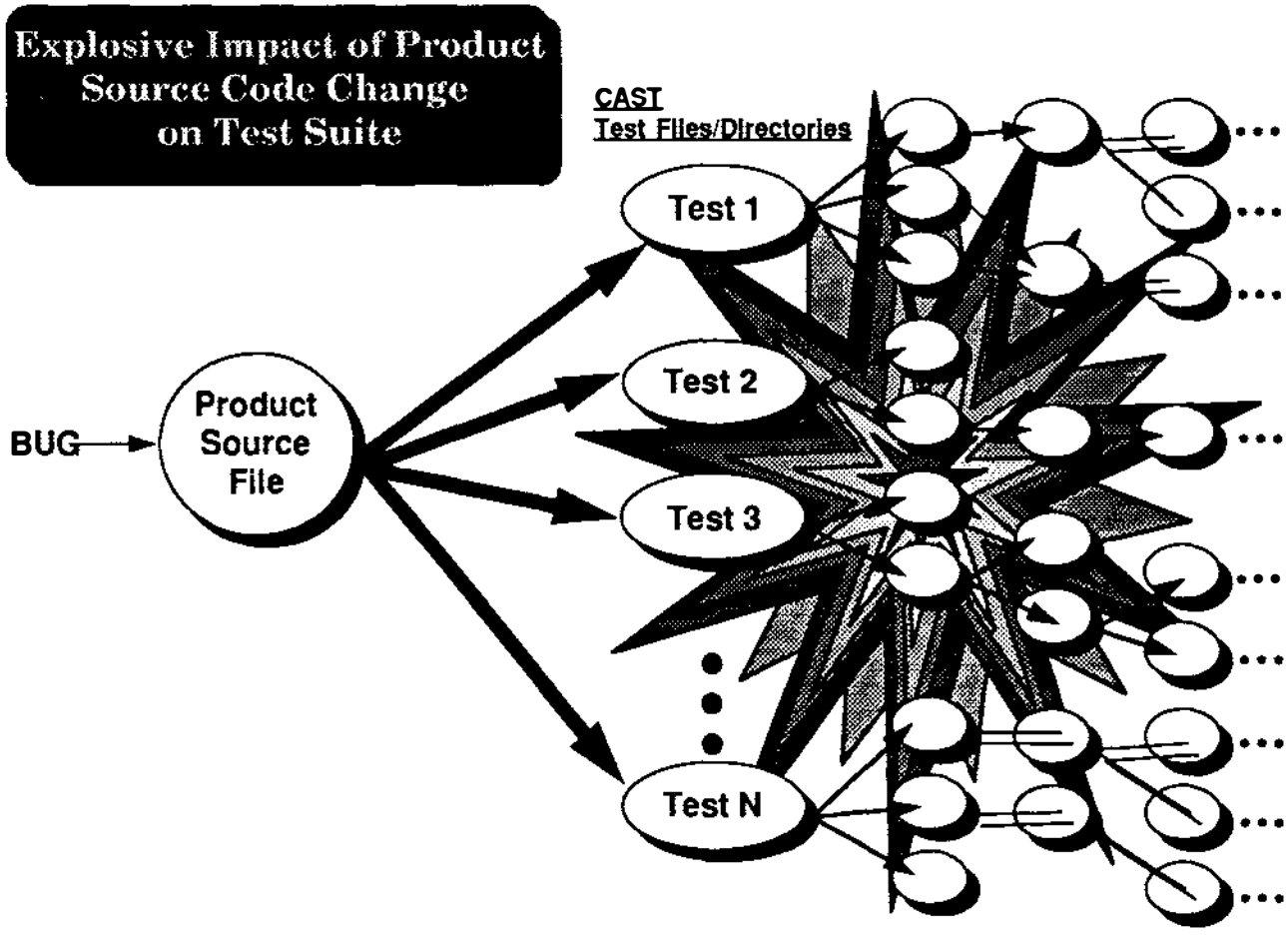
Today, we believe that over 200 AT&T/NCR projects are using some version of BUSTER for their testing. However, to our knowledge, few BUSTER customers (less than ten) have attempted the joint use of BUSTER and SCCS/RCS. I am aware of no group using SCCS/RCS without BUSTER at the present time. This leads to the belief that the additional control benefit is not worth the perceived high cost. The combination of Test Control and SCCS/RCS is most likely to happen, if our experience is any indication, when there are few tests, and the product being tested is mature.

6. A CASE IN POINT

We use BUSTER to test another UNIX system software product we develop called *The QUARTZ Remote Terminal Emulator.* QUARTZ runs on its own (i.e. a separate) computer that is electrically connected to the System Under Test (SUT). QUARTZ generates realistic loads by emulating users running applications at their terminals. An important feature of the QUARTZ software is that it accurately measures response times for all users.

We use the AT&T Sablime (an SCCS-based) system for configuration management and source control of QUARTZ. There are 230 C source and header files and a handful of directories in the QUARTZ SCCS source tree. Contrast this with our CAST suite. There are more than 500 BUSTER tests in our automated test suite.

It turns out that these 500 QUARTZ tests have more than 6000 files and directories! *So, for every QUARTZ source file to be controlled, there are, on average, two directories and more than 25 files in the test suite that have to be tracked.*

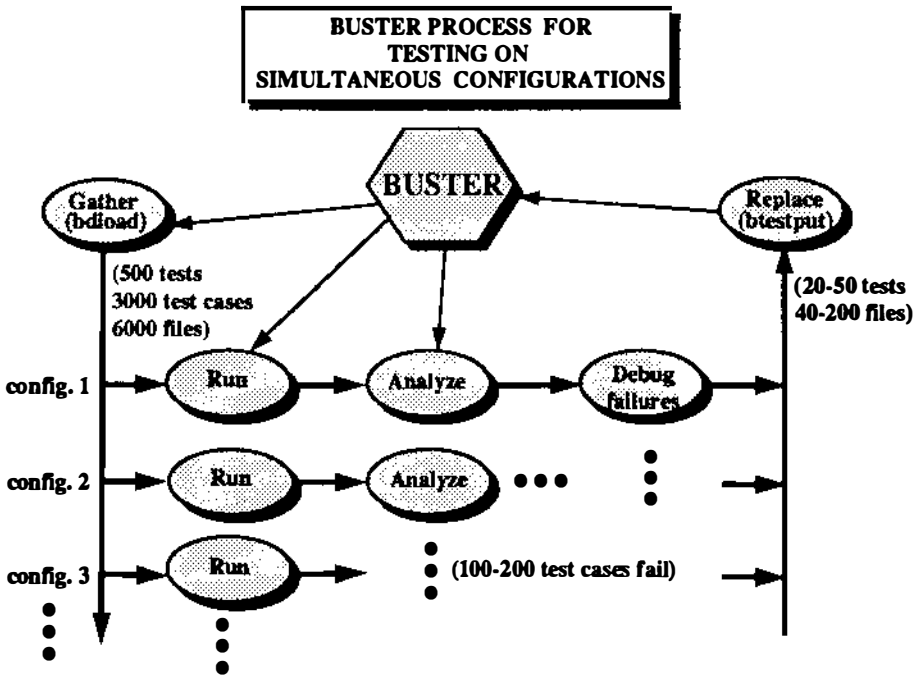


Our 230 product source files are easily managed with Sablime. QUARTZ is mature, and the directory and file structures are fairly stable. However, the underlying SCCS administration breaks down because our modest 500 test CAST suite contains so many unstable files and directories.

Our development experience can illustrate the problem. A QUARTZ software delivery (or load), on average, contains 20-25 Sablime Modification Requests (MRs), i.e. changes to the product.

These are usually bug fixes but may be new features. During system testing, we run the 500 QUARTZ tests simultaneously across 4 different configurations.

The testing staff (for both QUARTZ and BUSTER) has ranged from 1 to 2 people. (Most of the test development comes from the software developers who are required to submit BUSTER tests with any product modifications.) Generally we test over 8-12 different configurations. Our experience has been that roughly 100-200 test cases (20-50 TESTS) routinely "fail" on one or more of the configurations during a typical load (group of MRs tested during the cycle) because of the source changes to the QUARTZ product.[1]

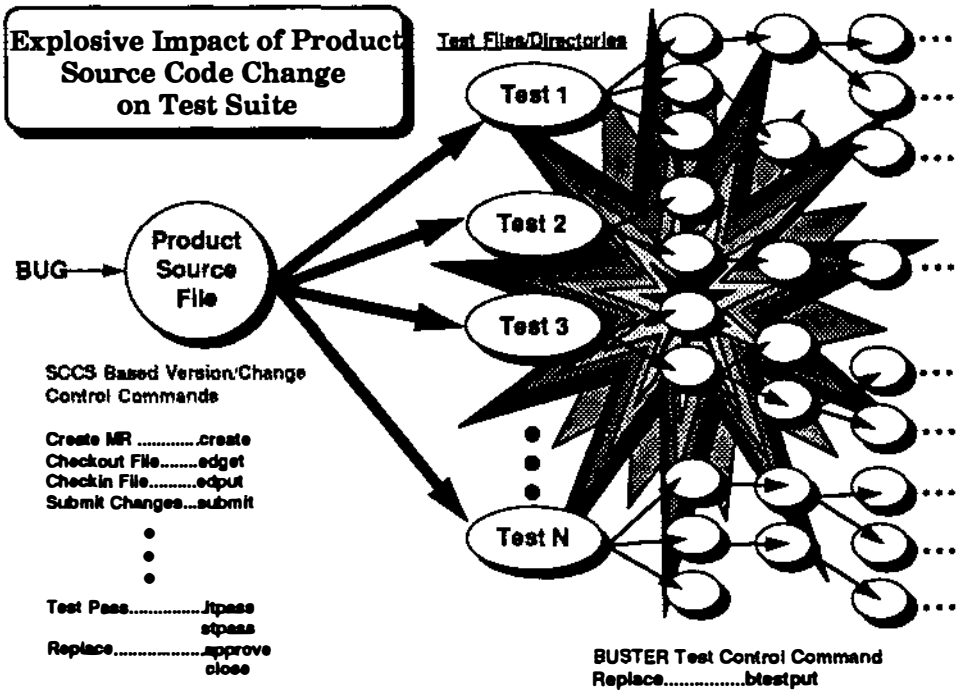


These test "failures" do not represent faulty behavior in the product and after analysis the tests are usually corrected. Although, we do not currently track individual file changes in the 20-50 tests that

1. And we try hard not make product changes that are likely to impact our test suite!

have to be updated, I think a safe estimate is somewhere between 40-200 individual file changes are made on average.

The reason it is not realistic to use a file-based mechanism for test control is simply that the overhead makes the cost far outweigh the benefit. For example, in our case, we would have to run several different user level commands (7 or 8 with Sablime) for every *file* that changed — even if we did somehow keep track on the target machine of only those files in tests that changed. Relatively minor changes to the CAST software would force us to run hundreds of user-level Sablime administration commands.



In contrast, the BUSTER **btestput** command can replace all files in the changed tests with a single command invocation (or at most the command is run once for each test update, i.e. 20-50 times.) In practice, there seems to be little penalty for this extremely simple and efficient mechanism[2].

7. THE PROMISED LAND: TEST LOCKING ON TARGETS

BUSTER Test Control reduces overhead in another way. Testers do not have to "check-out/check-in" a test. This is another idea that is hard for anyone to accept who is familiar with ordinary configuration management techniques. But test locking is arguably of questionable benefit for tests, even if changes are made on machines that have access to the SCCS/RCS database.

As mentioned, test failures are more often the result of test problems. These problems are usually noticed on target machines *after* the official tests have been downloaded. Test locking (i.e. check-out/check-in) at this point, *especially on individual files,* provides little additional benefit. It is not realistic to try and "lock" the test on all *target* machines where the official version has been copied.

Locking the configuration management database back on the *host* machine only works if the testers are required to "check" the *host* database locks before proceeding with changes on the *target*. This is hardly practical because test "debugging" first involves assuring that the product software under test is functioning correctly (before assuming the test is functioning incorrectly). It does not make sense to lock a test before determining that the test is indeed broken, and the test is often changed before this determination can be made (e.g. to emit more verbose output).

The solution using BUSTER Test Control mechanisms, in lieu of locking, is simply to assign test debugging responsibility to testers on the basis of tests — and *not* on the basis of configurations. As long as one person "owns" the debugging activity for a test that fails, test locking does not become an issue.

The BUSTER Test Control mechanisms notice tests that change. If the same test breaks on different configurations for different reasons, the tester will have to transport and test the "corrected" test on these configurations. After the test has been debugged, the tester simply replaces the official version on the HOST machine. A

---

2. I am only aware of one instance, over 8 years, that we used backup tapes to restore a test that we incorrectly updated.

single command has to be learned to replace tests. The BUSTER *replace* (**btestput**) command is run, at most, once per test change, rather than having to learn and run *seven or eight* different user-level commands for every *file* that changes.

Test Control is effective because both the product and testing software are improving together. Official copies of tests that "break" are changed as necessary, and retested when the next load arrives. This works as long as official versions of tests are always downloaded and used for testing. A simple database query can be used to download only those tests that changed since the last download. Regular management review of BUSTER reports can assure that only "official" test results are accepted.

8. CAVEAT

The BUSTER Test Manager, based upon Test Control, has proven to be a workable alternative to ordinary SCCS/RCS configuration management. BUSTER was designed and developed by end-users, e.g. system testers, and it is notable that the basic architecture has not changed since the original 1985 concept design. The design stressed maximum flexibility. It is easy to configure the BUSTER software for the needs and requirements of individual projects. Perhaps this flexibility — trying to be everything to everyone — is a weakness. Although many projects want to do things there own way, a lot of projects would be happier if BUSTER predefined more standards and then more stringently enforced them. A less flexible BUSTER system would require less on-site customization.

9. CONCLUSION

It is hoped the reader has gained a new perspective on the control of automated testing software. The reasons the obvious solution to formal control of CAST, e.g. ordinary SCCS/RCS change control systems, have not been successful are subtle. Although the SCCS/RCS mechanisms work well for product source control, they become very expensive when applied to testing software.

The idea that CAST software should be controlled stringently has merit, for a lack of adequate control leads to CAST errors which, in

turn, lead to undetected product faults. But file-based CAST software control with version control and file-locking soon becomes much more expensive to administer than the product source being tested. Previously, the lack of an effective low-cost alternative to SCCS/RCS file-based tools left much test automation software without any formal control.

The concept of *Test Control* will lower costs, and thus be superior to ordinary version and source control methods for the day-to-day management of test automation software under the following conditions: when the product under test is not fully mature, when the expected CAST software is likely to be large and prone to change, and especially when you do not have general access to the configuration management database on target machines.

Automated testing can dramatically improve quality and reduce overall development costs because many more tests can be run and analyzed than with manual methods. In practice, the automated test suite embodies product requirements and becomes a welcome additional stabilizing force for "change control." Now, quality managers have an alternative means of formal control. They can determine the optimal control mechanism for their CAST testing software by taking into account the sometimes subtle, but important differences between their product and CAST source code.

REFERENCES

[1] Beizer, B. Software Test Automation Issues. Software Testing Times, Volume 1, Number 2. Washington D.C., 1992.

[2] ANSI IEEE STD 829-1983, IEEE Standard Software for Software Test Documentation, IEEE Computer Society Press, Worldway Postal Center, Los Angeles, CA 90080.

Experiences in Applying Statistical Testing to a Real-Time, Embedded Software System

Kaushal Agrawal
and
James A. Whittaker

Abstract This paper describes a real application of statistical software testing conducted at the IBM (AdStaR) laboratory in Tucson, Arizona. The application is embedded microcode running in real-time which controls a high-performance tape storage system. Statistical test cases are generated based on a Markov chain description of the input domain of the microcode. The Markov chain accurately captures the relationship of input domain elements and supports analysis about usage probability distributions. We describe in detail the application environment and construction of the Markov chain. Further, we explore some standard computations on Markov chains and relate those that we found particularly insightful. We also describe the application of the test and the collection of software quality measures computed from a Markov chain-based software reliability model.

Biography of Authors:

Kaushal Agrawal is the Tape Products Test Coordinator at the AdStaR laboratory in Tucson, Arizona. He specializes in the use of statistical testing and analysis techniques for measuring the quality of software components in tape storage systems. Mr. Agrawal received the BSEE degree from the University of Arizona in 1990. Mr. Agrawal's current address is 9000 South Rita Road, Dept. 79F/041, Tucson, AZ, 85744 or *kagrawal@vnet.ibm.com.*

Dr. James A. Whittaker is a Computer Scientist at Software Engineering Technology. Dr. Whittaker's research interests are in methodical and statistical testing techniques and reliability estimation for software. He received the BA degree in Computer Science/Mathematics from Bellarmine College in 1987 and the MS and Ph.D. degrees in Computer Science from the University of Tennessee in 1990 and 1992, respectively. Dr. Whittaker's current address is 601 South Concord Street, Suite LLC, Knoxville, TN, 37919 or *whittake@cs.utk.edu*.

1. A Finite-State Model for Statistical Software Testing

The term *statistical software testing* refers to a general technique of applying external stimuli to a software system based on a probability distribution that describes some specific user or class of users. Statistical tests are generally a black box activity, meaning that the tests are designed from the specification and not from a clear box view of the implementation.

Statistical testing for software is often described as a sampling procedure similar to "drawing balls from an urn". In this case a "ball" is a user input selected at random (normally with replacement), according to a predefined probability distribution, and applied to the software; causing another input to be selected and the process repeated. The difficulty with sampling for software is that each time an input is drawn, the probability distribution of inputs is subject to change. This is because state data builds as software is executed and this "saved-state" affects software behavior and the input probability distribution for users. Thus, a more accurate model must be capable of describing software usage in terms of a set of conditional probabilities; modeling the input probability distribution conditioned on prior inputs.

In computer science, such situations are often treated as sequence generation problems. We are interested in generating a *sequence of inputs* that constitutes a typical execution of a software system. Models for sequence generators can take any number of forms. For sets of sequences that constitute a *regular formal language*, regular expressions, regular grammars, and finite-state machines (FSM) are the pertinent models. *Context-free languages* require the more complex context-free grammar (CFG); however, FSM's do capture some properties of context-free languages as long as the depth of recursion, inherent to such languages, is limited to finite size. We could go further in the language hierarchy to context-sensitive languages and beyond; but we move further away from models for which significant computation and analysis can be performed. As will be shown in this paper, such analysis lends useful insight into the testing process and justifies the use of FSM-based models. Furthermore, we have found the simpler models to possess sufficient power to model many aspects of usage of complex software systems, including the embedded application described in this paper.

This paper describes an application of statistical testing where an FSM model is used to control the sampling process. The state set of the model is the input domain of the software system under test. The directed arcs connect related states in an order such that all possible input sequences can be obtained by simulating paths of the FSM. Probabilities are attached to the arcs so that these paths can be generated randomly, creating a basis for the statistical test. Specifically, a probability distribution is assigned to the exit arcs from each state. Assigning probabilities in this manner effectively converts the FSM to a *finite-state, discrete-parameter Markov chain.*[1] This is an important observation because Markov chains are highly tractable stochastic processes, enabling significant analysis about the accuracy of the probability distributions and expectations about test time and budget. Furthermore, they allow the use of a new method of reliability modeling based on Markov chain generated test data.

The Markov chain captures the set of conditional probability distributions that define typical use in a concise statistical framework. The chain can be constructed at varying degrees of abstraction so that even

[1]Readers unfamiliar with Markov chains and the associated theory may find the following visualization helpful. A Markov chain can be pictured as a state machine in which transfer of control (i.e., state transition arcs) occurs according to a specific probability distribution. These *transition probabilities* are normally shown as labels attached to the arcs of the machine. See [Feller, 1950] for a general treatment of Markov chains.

very large systems can be accurately modeled. The benefits of using such a model are the precision it lends to the sampling process and the opportunity for controlling and analyzing various aspects of the testing process. Testing with such a model becomes a scientific process of measurement and management instead of the ad hoc application of crafted test cases. A formal treatment of Markov chain usage models can be found in [Whittaker *et al*, 1993].

Markov chain usage models (called usage chains) consist of a *structural* component which captures the externally-visible state behavior of a software system from a user's perspective and a *statistical* component that describes the set of conditional probability distributions with which users apply inputs. Test input sequences can then be generated from the chain that form the basis for a statistical certification of the software's quality. The more detailed the structural component, the more likely the certification results will apply to actual use of the system. Similarly, the more accurate the statistical component, the more likely the certification results will correspond to real world use by typical users.

In this project at AdStaR, we chose an implementation strategy of Markov chains that we felt was a serious effort at full implementation, yet modest enough to keep the test team fully aware of the entire intellectual process. The application documented in this paper is not an attempt at a religiously conducted statistical experiment within the Markov chain framework. Instead, it is the first step toward a full implementation of Markov chains leading to accurate certification of the software products developed at AdStaR. In other words, we decided to start small and master each technique in a step-wise fashion. In this manner, we hope to maximize team understanding of statistical certification testing and to develop and tailor each component to our particular application environment. This being our goal, we began with the main focus on the structural component of the Markov chain in order to get the best possible usage description of our system. Our attention to the statistical component is admittedly shallow and we interpret our certification results with this in mind. However, we have gained valuable experience in interpreting and finding uses for the statistical measures that we did compute. As yet, these metrics are applied to our development practices only. As our expertise in usage modeling grows, we are able to concentrate more on obtaining accurate probabilities and to work towards attaching quality measures to actual products.

In the next section, we describe the application environment at AdStaR and give some project statistics that are of interest. Section 3 defines the Markov chain model for the application and shows how it was constructed and analyzed. Section 4 describes how the test cases were assembled from the model and executed against the microcode. We compute a number of software quality measures and show how they were used to make stopping decisions and to estimate the quality of the current software version. The final section concludes our experiences and relates future directions in statistical software testing at the AdStaR laboratory.

2. The Application Environment

The software system that was used for this project is a real-time system which controls a high performance dual tape drive. The existing 3490E Model C11 and C22 is IBM's mid-range tape drive with ESCON (Enterprise System CONnection) fiber optic serial interface and the System/370[2] parallel interface. The current development on this product is to offer an additional interface for the existing tape drive,

[2]3490E, ESCON, and System 370 are registered trademarks of the IBM Corporation.

specifically, the SCSI-2 (Small Computer Systems Interface) differential, fast, and wide interface.[3] This new interface complies with the SCSI-2 ANSI specification and must attach to any host computer which contains a SCSI-2 host adapter card. As an IBM product, the testing and development of the SCSI-2 interface microcode is geared toward the RS/6000 work station as the host attachment. Figure 1 depicts the system under test connected to the SCSI-2 host (via the SCSI-2 bus) and the control unit.

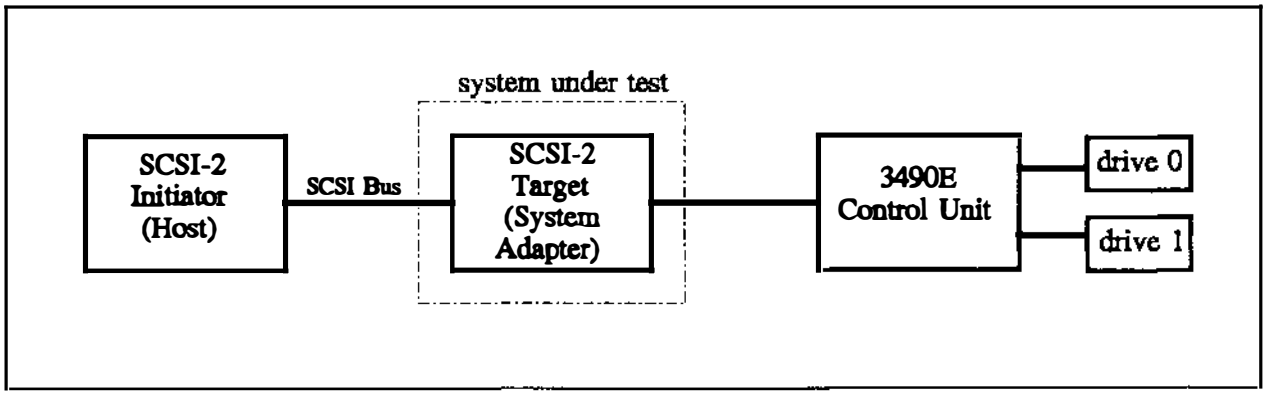


Figure 1: Diagram of the System Under Test

The SCSI-2 interface card contains Intel's 80960 (I960) microprocessor which allows the software to directly communicate with other hardware on the card including the SCSI-2 chip (third generation level). This card also contains modules to interface to an existing control unit which communicates with the physical tape drive. A high-level view of the SCSI-2 protocol is that it consists of two separate layers. First, a *protocol layer* is responsible for establishing communication between an initiator device and a target device and controlling bus contention from multiple initiators and targets. Second, a *command layer* passes commands to the target. In this case, commands are for information dumps to the tape drive (i.e., backup commands), information retrieval (i.e., restore commands), and transaction processing (e.g., sorting, searching, etc.). The microcode for the system adapter has the responsibility of sending and receiving commands, via the SCSI-2 bus, to and from any host attachment and to instruct the control unit to manage the tape drive accordingly.

The engineering team for this project consists of five developers and four testers. The development team employed the C programming language for all microcode development for the system adapter. The code was engineered using the Cleanroom methodology [Mills *et al*, 1987]. The code consisted of approximately 86 KLOC of design, of which about 22 KLOC was actual C source code.

The test team used multiple strategies for testing the software. The bulk of testing was performed statistically and was supplemented by crafted test cases in the early increments; however, as experience (and confidence) was gained in the modeling techniques most test case crafting was stopped in favor of additional statistical tests. Some acceptance and verification tests, required by AdStaR and its customers, were also performed. Included in this set were tests which simulate various DC power failures and environmental factors such as high humidity, temperature, etc.

[3]The differential option specifies distance whereas the fast and wide options allow a data transfer rate of up to 20 megabytes per second.

The system adapter microcode was developed in increments using the Cleanroom methodology. An increment represents a end-to-end slice of a software system that is externally executable and verifiable. Thus, an increment can be tested from an external, or black box, viewpoint. Prior increments are included as a part of newer increments. This particular product has currently undergone 3 such increments of development and testing.

The increments were determined by selecting a subset of SCSI-2 commands and protocol that the system will accept. The first increment contained three basic SCSI-2 commands (Test Unit Ready, Inquiry, and Request Sense), and the initialization of the hardware. These commands do not use the actual tape unit but do cause two-way interaction with the SCSI-2 bus and, therefore, are externally visible. Only the parts of the protocol necessary to execute these commands were implemented in the first increment. The second increment contained the entire SCSI-2 protocol and three additional commands which allow data to be written to and retrieved from the tape drive. The third increment provided the remaining SCSI-2 commands, some of which allow the host to modify drive parameters for some applications.

3. Construction of the Markov Chain

The first choice faced in building a usage model is to decide who the user of the system is. In this project, the choices are several. We could define the user to be (i) the human user of the host, in which case inputs would be keystrokes to an application which uses the SCSI-2 card, or (ii) the applications on the host, in which case the inputs would be the application-level protocol to the SCSI-2 adapter card, or, (iii) the SCSI-2 bus, in which case the inputs are SCSI-2 commands and protocol. Our decision reflected a study of the complexity of each as well as the thoroughness of the resulting tests. We selected the third option, the SCSI-2 bus, as the user because we felt that its inputs were more completely defined (via the SCSI-2 ANSI specification) than the others and that we could achieve the high level effects of both applications and the human user at the bus level. However, our analysis of probability distributions spanned all three levels of system use.

Use of the system adapter consists of sequences of SCSI-2 commands and protocol messages. Such sequences are of the form (i) invoke, i.e., establish the desire to communicate with the target drive, (ii) setup initial communication parameters, (iii) begin communication through input/output processes (IOP), and (iv) terminate communication. This process is shown in the upper portion of Figure 2 as a four state machine. The loop back from "Terminate" to "Invoke" denotes the transition to a new usage sequence. The shaded states "Initial Setup" and "Input/Output Processing" are actually non-terminal states that represent more complex sub-processes. As shown in the lower portion of Figure 2, the "Input/Output Processing" state is expanded into a five state machine that gives a more detailed description of that state.

Each shaded non-terminal state in Figure 2 is expanded into a lower level state machine which may also contain non-terminal states but exists at a more detailed level than its parent state. This scheme was adopted to facilitate review of the model with a diverse set of project stakeholders, including users, interested in high-level concepts, and test team personnel who require more specific details. Further, it allowed the opportunity to divide the workload associated with model construction and to maintain a large chain that might otherwise have been unwieldy. Only small steps were made at each layer of abstraction and each step was reviewed by the team before continuing. Eventually, a layer was reached which contained only terminal states and thus concluded our FSM model. Figures 3 and 4 depict this situation for one specific path through the model. In Figure 3, the "Standard Sequence" state is expanded to its corresponding state machine and subsequently shows the expansion of the non-terminal state "Command

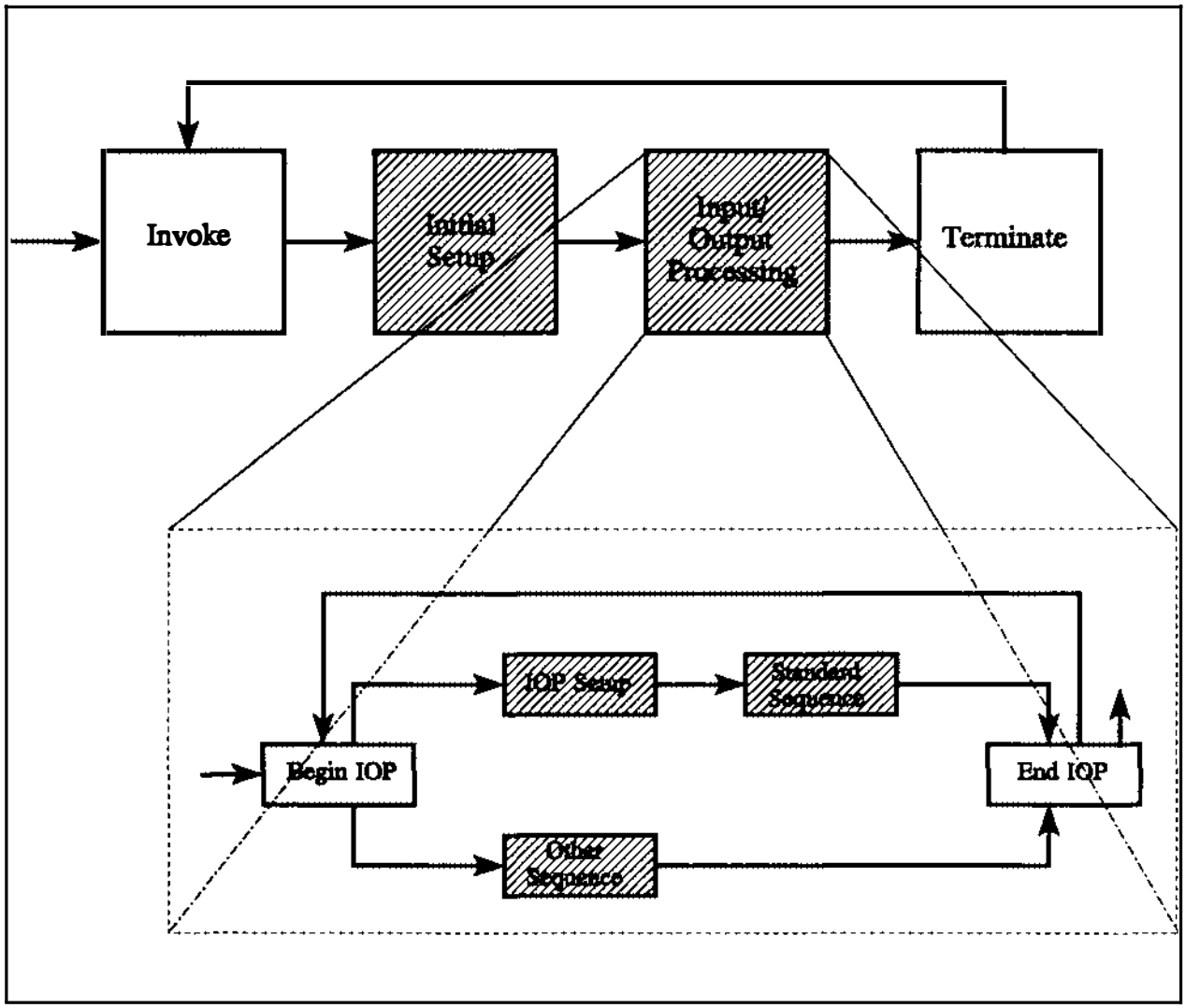


Figure 2: The First Layer of the SCSI-2 Usage Model and the IOP State Expansion

Sequence". Figure 4 shows the state machine for "Write Command" which contains only terminal states and is at the lowest level of detail.

It was sometimes the case that decisions were made to leave some states as non-terminals because it seemed that additional detail yielded no substantial payoff in terms of test effectiveness. In fact, after the second increment, we reversed the state expansion process for much of the SCSI-2 protocol states because team consensus was that the concentration should be on the SCSI-2 commands (where the real usage possibilities exist) and that the protocol was thoroughly tested in the first two increments.

The state machine for each non-terminal state has an initial state and a final state so that the entire diagram can be substituted for its parent state. Making such substitutions for each non-terminal state results in a state machine with 49 states for the first increment (increment 0) and 381 states for increment 1. For increment 2, where much of the protocol was kept as non-terminals, the resulting model was a much smaller 132 states. The specifics of each model are summarized in Table 1.

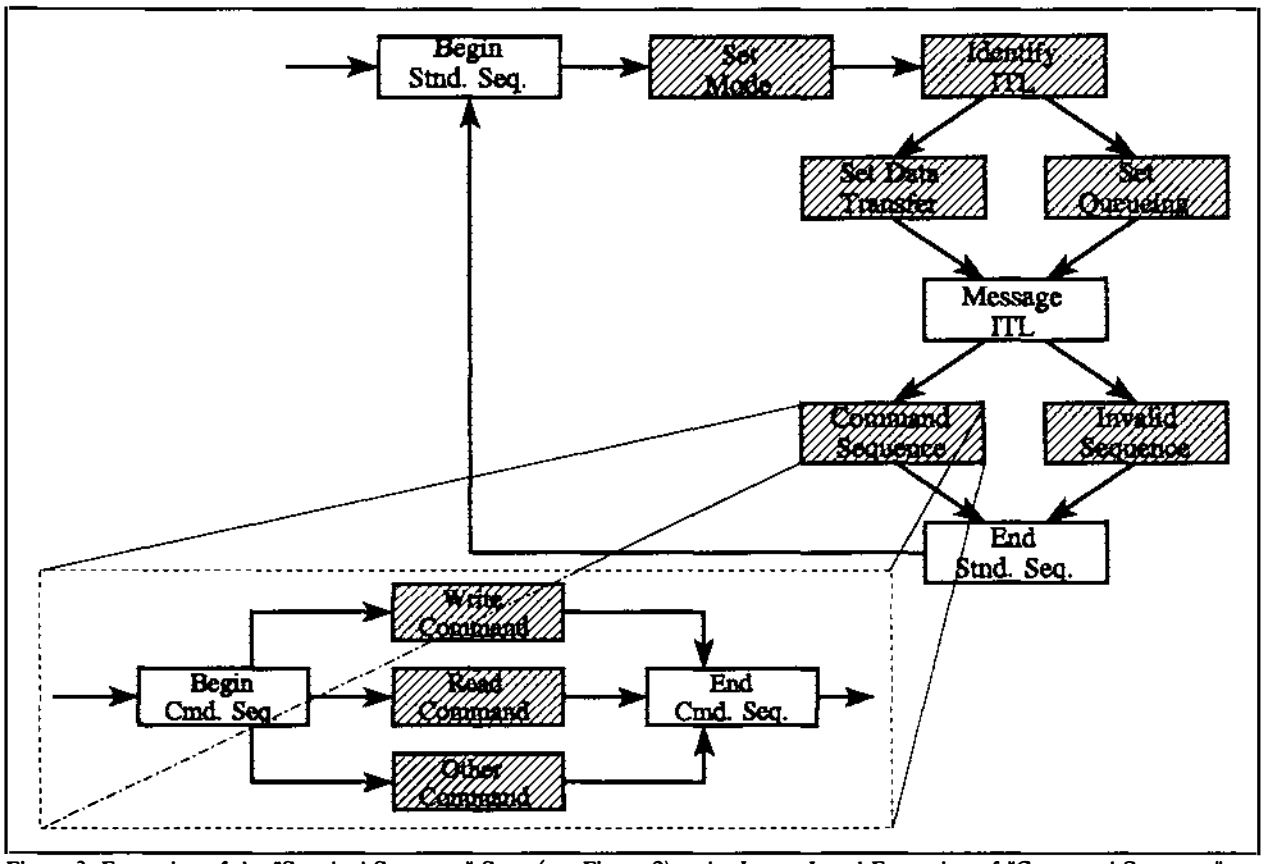


Figure 3: Expansion of the "Standard Sequence" State (see Figure 2) and a Lower-Level Expansion of "Command Sequence"

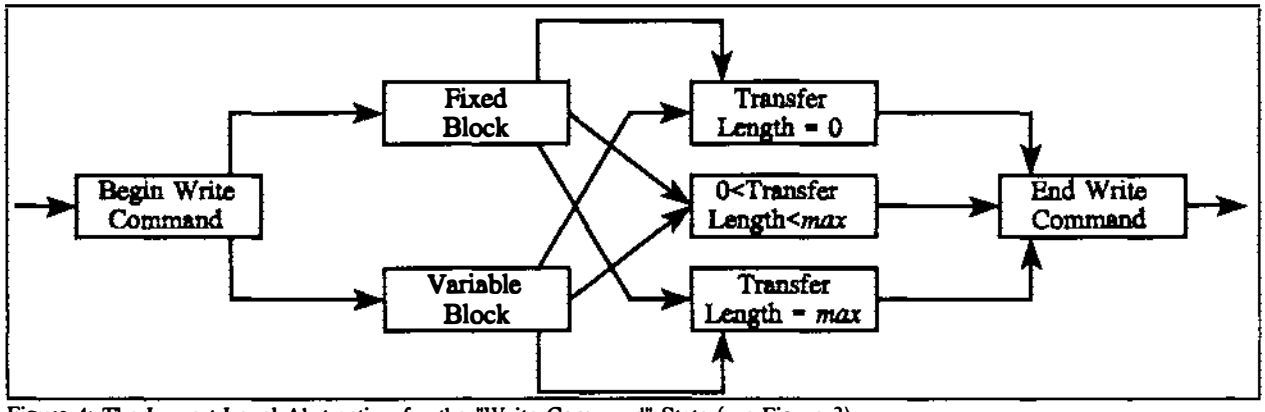


Figure 4: The Lowest Level Abstraction for the "Write Command" State (see Figure 3)

Table 1: Model Information by Increment				
Increment	No. Of States	No. of Arcs	Probability Distributions	Time to Develop[4]
0	49	120	uniform	10 days
1	381	639	intended	45 days
2	132	209	intended	30 days

As a final note on the construction of the usage model, we add that several times during model development we discovered input history dependencies that necessitated that we maintain duplicate blocks of states in the model to account for different uses of the same command. For example, commands can be sent either linked or unlinked which causes particular fields to be set to a certain value. In this instance, two copies of each command (i.e., the states and arcs that model the command) were included in the model to describe the linked and unlinked conditions. In this manner, we were able to model any depth of data dependencies at a cost of increased model size.

The intended probabilities (with the exception of increment 0) were established based on hypothesized customer use. For increment 0, the commands possible do not form actual user sequences, therefore, no data exists to use for probability estimates. Thus, the most reasonable guess seemed to be uniform distributions across the exit arcs at each state. For increments 1 and 2, real customer data was available. The system under test exists for the ESCON and System/370 interfaces; thus, we monitored existing customer tape drives to establish a basis for probability estimates. However, in all honesty, we did take liberties with guessing about particular events which did not occur in our customer data. One area that we seek to improve in our current and future projects is the method of assigning transition probabilities.

After initial probability estimates were established, we used a prototype CASE tool called SUMIT (Software Usage Modeling and Integrated Testing) to analyze the resulting Markov chain. The tool implements the mathematics described in [Whittaker, 1992] (which is summarized in appendix A) to produce an analysis of each state according to its (i) steady-state probability, (ii) probability of occurrence in a single sequence, and (iii) expected time to occurrence. This information was of paramount importance in determining the overall effect of the assigned probabilities and in predicting testing costs. For example, Table 2 shows some summary statistics produced by the tool that give insight into the expected amount of testing one must perform until each state and arc of the model appears in the test sequences.

[4]No precise records were kept for project effort or duration. This data represents approximate effort and comes from estimates made by the team leader.

Table 2: Markov Analysis Summary Statistics			
Increment	Expected Sequence Length	Full State Coverage Expected at Sequence Number	Full Arc Coverage Expected at Sequence Number
0	41	40	108
1	70	463	466
2	951	74	843

Our main use of the analytical results was in two areas. The first use was as an indication of how close the probability assignments were to typical customer usage. We compared, for example, the expected sequence length from the model to the average sequence length of the captured usage sequences and adjusted probabilities to bring the two in-line. This ability to perform analysis of probability estimates eased our conscience a great deal concerning the transition probabilities for which we had no data. Our procedure for analysis was as follows. We established "usage policies", e.g., one policy was that information dumps were 20 times more likely than information retrievals, that defined high-level use of the software. We then verified through the analysis provided by SUMIT that each policy was supported in the model. It often took several iterations of parameter adjustment and re-adjustment before all the policies were in effect. Our goal was to define a set of policies that represented all the concrete information that we had on how this device will be used in the field. It is our intention in the future to involve the customers with helping us establish policies and determining whether or not the policies represent typical use.

The second use of the analytical results was as a means of planning the duration of certification testing and to estimate testing costs. Our goal for the final version of an increment was to run enough sequences to traverse every state and arc while observing no failures. Thus the last two columns of Table 2 gave us an indication of the expected number of sequences that must be executed to achieve this goal. One must keep in mind that these results are expectations only. Whenever one is dealing with probabilities there are rarely absolutes. However, in practice we found these estimates to be accurate enough to use as the basis for planning schedules and estimating the costs associated with executing a set of test sequences.

4. Execution of the Statistical Test

Once a model was obtained that the team agreed was a good representation of customer use and also a practical model with which to conduct testing, we proceeded with the execution of the statistical test. The first step in performing the statistical test involves simulation on the usage Markov chain to obtain input sequences. A path through the chain from the "invoke" state to the "terminate" state represents a single use of the system adapter. These sequences can be generated using a random number generator and simulating paths of the chain according to the transition probabilities. We used SUMIT to generate these sequences automatically.

The generic input sequences were converted into an executable format then compiled and executed against the system. For each stimulus in the Markov chain, we developed a C code block that (i) encoded the stimulus as an executable C code segment and (ii) encoded the expected response when the stimulus is

applied.[5] We called this part the "transform" because it effectively transformed the generic stimuli sequences into an executable format. We built two transforms for each input for existing IBM execution platforms.[6] The test case creation process is summarized in Figure 5.

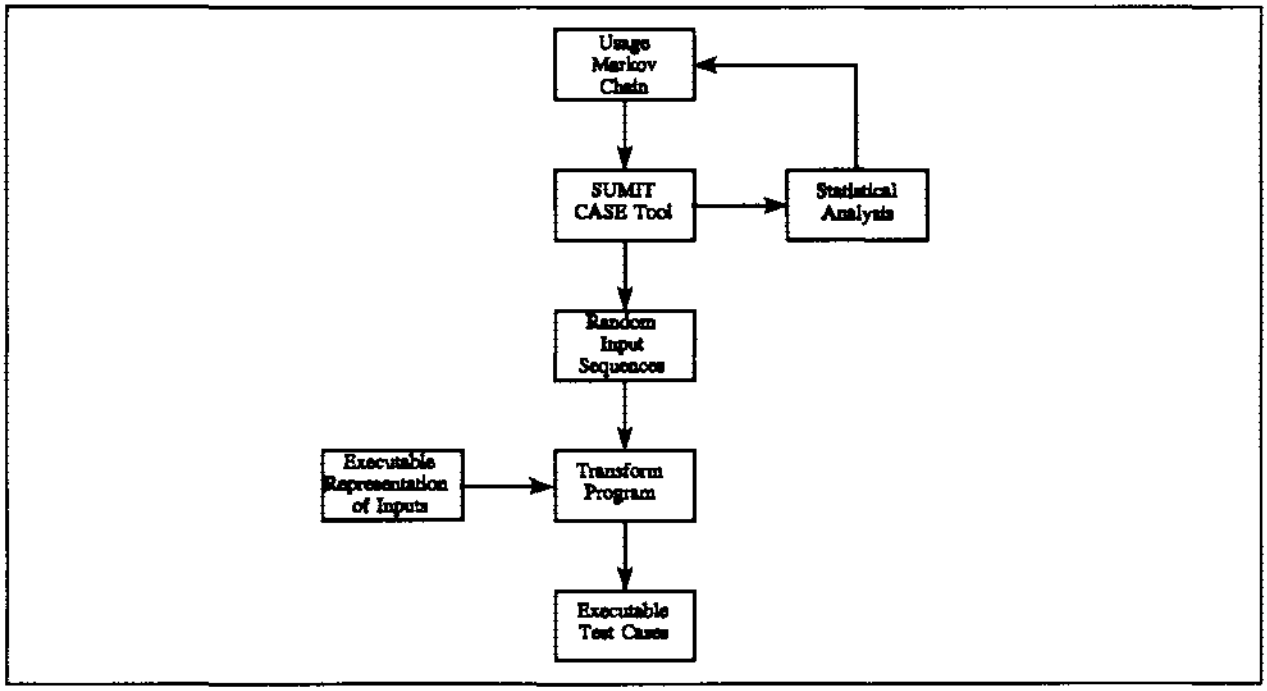


Figure 5: The Test Case Generation Process

Response checking was not completely automated. There were several instances of stimuli and combinations of stimuli that could only be checked manually. In these cases, the transforms were responsible for recording specific bus transactions so that we would be able to perform post-analysis of the execution results and determine whether the software performed as specified.

As the test cases were executed, the results were input to the SUMIT tool which implements a software reliability model based on the execution results organized as a Markov chain. This *testing Markov chain* models each sequence applied to the software as well as failure data observed during testing. A testing chain has the same structure as the usage chain plus additional states and arcs to account for each individual failure located during testing. Failure states are included in the testing chain in the exact location that the corresponding failure occurred within a sequence. The testing chain's transition probabilities are established using the frequency counts of the state transitions of the actual sequences applied to the system.

Testing chains make it possible to compute many software quality measures without making assumptions

[5]Not all responses could be directly encoded into the sequences. Some were manually inserted after the sequences were generated.

[6]The execution platforms used were COMET and CBASE which are IBM internal use platforms for testing SCSI-2 channel adapters.

about the underlying distribution of failure appearance rates. Because the test results are based on data generated by a Markov chain, many important distributions within the data are computable. This fact allows the computation of many standard software quality measures such as reliability, MTBF, etc. The mathematics about testing chains is summarized in appendix A. For specific details about testing chains and their construction see [Whittaker, 1992].

The analysis was aimed at organizing data to make specific decisions during the testing process. For example, as failures are found, one important decision is when to stop testing and submit the code for re-engineering. Furthermore, we needed to decide whether testing should continue during re-engineering or halt to wait for the code fixes. Our policy was to continue testing only when our statistical estimations were showing quality gains despite the failures. That is, as long as the reliability was growing, we would continue to test on the current version while the development team prepared the new version. Our analysis centered around two important areas of decision making: (i) decisions concerning intermediate software versions and (ii) decisions concerning the final version that will be released to users.

We constructed a separate testing chain for each version of the software as testing progressed. In other words, as code fixes were made to correct software errors, the testing chain was re-initialized so that it modeled only the current version of the software as it was submitted for testing. Once any change was made to the software, a new testing chain was constructed. Our goal was to obtain a testing chain for the last version of the software which contained no failure states. Thus, the customer will receive software with no known errors.

These testing chains are a source for significant analysis about the testing process. They yield data that aids in determining the quality of a software system as well as data that supports stopping decisions. Computation using testing chains is performed based on the relationship of the chain's failure states, installed when failures are observed, and the usage states, which are inherited from the usage chain. Although our experience base is limited to this single project, we are beginning to notice patterns with the data from testing chains. For example, we established an unofficial ".1 threshold" for the $d(U,T)$ measure (explained below) within the test group because we came to associate this distance level with our own subjective notion of high quality.

Among the items computed during this project were (i) the reliability, R, (the probability of a failure-free execution), (ii) the discriminant function (the stochastic difference between the usage chain and testing chain), and (iii) the distance function (the linear distance between the testing chain and usage chain when interpreted as vectors in multi-dimensional Euclidean space). The first measure is a probability-weighted reliability calculation; meaning that each failure is probability-weighted according to its location within the usage structure. Failures which occur on high probability paths are more heavily weighted than failures on low probability paths. Thus, each failure will impact the reliability according to its probability of occurrence (which is computed analytically from the testing chain). The next two functions are measurements of the progress of testing related to the representativeness of the sequences executed on the software. Since the testing chain is a model of the testing process up to the current sequence applied, i.e., a model of what *actually* occurred in the test (including failures), and the usage chain is a model of what *should* happen then a measure of the difference between the two constitutes a measure of testing progress. For more about the derivation and justification of these measures see [Whittaker, 1992].

An execution report appears in Table 3 for the first 20 sequences of version 1 for increment 1. These reports were generated for each version of the software. We added failure states to the testing chain each time we encountered a new failure and updated frequency counts when failures were encountered multiple

times. The reappearance of a previously detected failure was usually obvious by its context within the sequence that caused its discovery; however, it sometimes took analysis from developers to confirm that a particular failure was a repeat. This gave us an accurate picture of the current quality of the software. As the report shows, failures were observed during execution of sequences 1, 12, 15, and 16. Notice the drop in the reliability, R, column as a result of these failures. On the sequences in which no failures were observed, the reliability rises proportional to the significance of the sequence; e.g., a long sequence or a high probability sequence causes a larger rise than a short, low probability sequence. Intuitively, the more information that a sequence supplies about the quality of the software, the more it impacts the quality measures. If the reliability growth remained flat for an extended period, then the software was given back to the development team for code fixes. At this time, testing could continue (using the existing code and testing chain) or stop. When the new version was delivered, a new testing chain was created and the new code was used. The discriminant function, denoted $D(U,T)$, and the distance function, denoted $d(U,T)$, were used in a similar manner to track the representativeness of the test suite. $D(U,T)$ is not computable in Table 3 because all the arcs are not traversed in the 20 sequences. However $d(U,T)$ shows typical behavior in that it rises sharply when failures are discovered and otherwise fluctuates according to specific sequences involved. Notice that a slight rise occurs at sequence 9 even though no failure occurred. This statistical fluctuation is common to such metrics. Intuitively, a sequence causing such a rise supports low frequency events in the model and thus makes the sample look less representative.

In addition to the analytical results, we tracked the percentage of states and arcs covered by the test sequences; as shown in the last two columns of Table 3. These measures give testers a second source of information about the progress of testing. Thus, testing progress was captured in two different criteria: (1) test representativeness and (2) model coverage. Both of these impacted stopping decisions.

Table 3: Certification Report for Increment 1 version 1

sequence number	Pass/ Fail	R	$D(U,T)$	$d(U,T)$	cumulative % states covered	cumulative % arcs covered
1	F	.3333333	∞	.95422462	15.74	10.01
2	P	.5714286	∞	.94096713	18.63	12.51
3	P	.6923077	∞	.92004155	21.78	15.80
4	P	.7804878	∞	.88836961	29.39	21.90
5	P	.8181818	∞	.88379349	29.92	22.53
6	P	.8450704	∞	.87106992	31.75	24.41
7	P	.8651685	∞	.86183782	33.33	25.82
8	P	.8807339	∞	.85858036	34.12	26.44
9	P	.8936170	∞	.85901543	35.95	28.01
10	P	.9039548	∞	.83103679	39.63	31.14
11	P	.9121951	∞	.82697595	40.15	31.76
12	F	.8437048	∞	.84027813	44.61	35.83
13	P	.8553459	∞	.81533023	46.19	37.55
14	P	.8651791	∞	.79547722	46.19	37.55
15	F	.8271236	∞	.80811357	54.59	44.13
16	F	.7873131	∞	.85846864	54.85	44.28
17	P	.7948111	∞	.82876494	58.26	47.26
18	P	.8055149	∞	.80653940	60.89	50.07
19	P	.8187006	∞	.74000197	61.94	51.48
20	P	.8274287	∞	.70925508	62.72	52.58

Table 4 summarizes the results for each increment's certification. Our policy was to require that the last version of an increment run failure-free (i.e., R=1.0) which did, indeed, occur and that every state and arc be traversed at least once. The latter requirement was met on each increment except 1 where we decided to stop without complete coverage because the team had gained sufficient confidence without it.

Table 4: Certification Results Summary

inc	ver	no. of tests	no. of failures	no. of failure instances	R	$D(U,T)$	$d(U,T)$	% states covered	% arcs covered	time (days)
0	1	300	4	35	.78	.01342	n/c	100%	100%	9
	2	1000	0	0	1.0	.00081	.02872	100%	100%	5
1	1	50	12	21	.67	n/a	.64268	76%	66%	10
	2	150	6	28	.84	n/a	.52986	88%	76%	5
	3	754	10	39	.96	n/a	.33595	99%	97%	28
	4	1000	0	0	1.0	n/a	.10863	99%	98%	13
2	1	155	8	110	.60	.00033	.27078	100%	100%	16
	2	250	5	91	.75	n/a	.24835	98%	96%	4
	3	490	0	0	1.0	.00001	.05225	100%	100%	18

5. Conclusions and Future Work

The usage modeling techniques described in this paper have significantly improved the focus and effectiveness of software test activity at AdStaR. Although the initial model development required training and concentrated effort, the payoff has made the investment worthwhile. The model makes the generation of any number of effective and informative test cases a simple matter instead of the time consuming process of test case crafting that marked the business-as-usual strategy. Perhaps the greatest achievement of the modeling process is the amount of understanding of the software's requirements and specification that it enforces. Each stimulus must be carefully considered for its effect under any possible software state in order for the model to be accurately constructed. Since all this knowledge is packaged into a model which is the basis for test case generation, the effectiveness and completeness of the test suite is ensured.

Since this is the first experience with statistical testing at AdStaR, we failed to take full advantage of the power of the underlying mathematics. We mainly concentrated on building good structural models and treated the probability assignments and analysis as of secondary importance. These latter characteristics are being given more serious effort on our current projects (three new projects have been initiated). We have begun to get customers involved in the model analysis so that it is more likely that the probability estimates are accurate. Also, as some team members have become proficient in the intricacies of Markov chain theory, we have improved our ability to form judgements and take actions based on the statistical analysis. Early on we relied on expert consultants for statistical and mathematical expertise; however, as

our confidence grows in our ability to build good models, our interest in the statistical aspect grows proportionally. Team members found, much to their surprise, that extensive experience in testing and aspects of testing made up for a lack of formal training in statistical theory. In other words, intimate knowledge of the application made interpreting the statistics much easier than was originally thought (or feared).

Current certification work includes two important improvements in the above techniques. First, we have decided to adopt a new labeling structure for the usage Markov chain which provides a separate vocabulary for the states and the arcs of the chain. Stimulus labels will be moved from the states to the arcs and the states will be labeled with the externally visible software state that is reached as a result of the current stimulus history as shown in Figure 6. Thus, we are directly modeling the fact that as stimuli

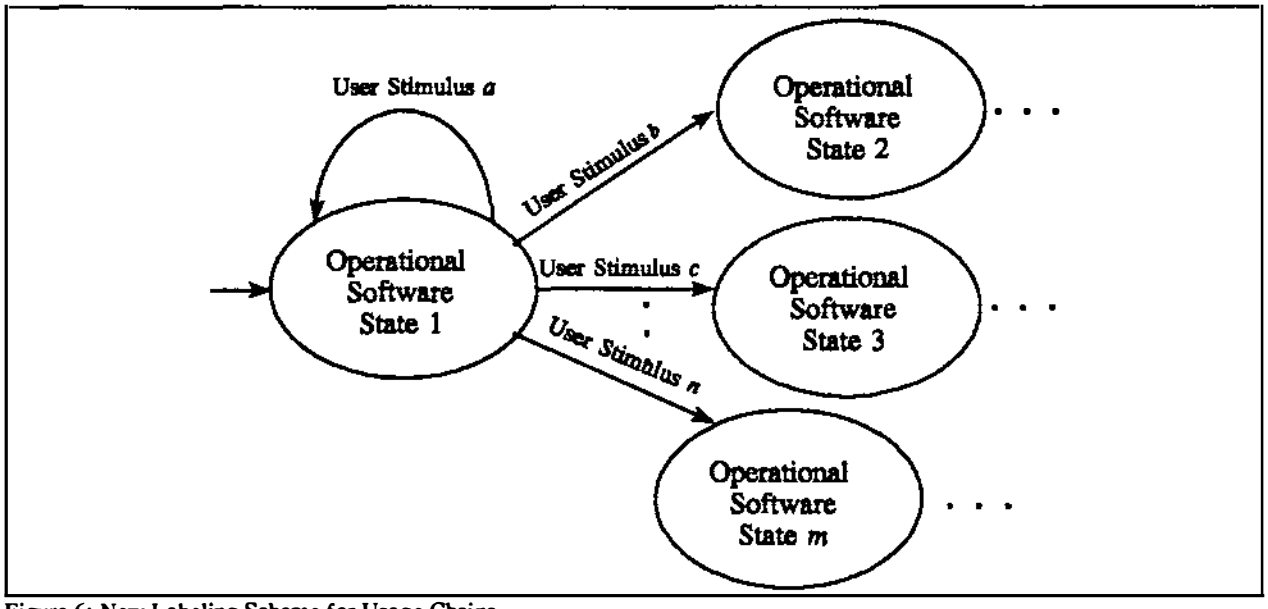


Figure 6: New Labeling Scheme for Usage Chains

are applied, the software changes state as result of the entire stimulus history. The effect is that the size of the models will be significantly reduced because of the additional power that this method gives. Second, we are experimenting with a policy-oriented approach to defining the transition probabilities. As stated earlier, we found that assigning individual transition probabilities was difficult. Even though this was counteracted somewhat by the analysis, it did slow down the model building process. However, we found that we were able to state a good number of high-level "usage policies" about specific user behavior. For example, a policy concerning customer usage may be that "transaction processing occurs in less than 1% of the uses of the software". When policies are formulated in this fashion, we can map the events (e.g., transaction processing) in the policy to specific states and arcs in the chain and define the probabilities so that we obtain the desired ratio (e.g., less than 0.01).

Acknowledgements

The authors thank the Design and Test System Tape Team, headed by J. Ryan, at AdStaR for their effort on this project. Special recognition goes to all the members of this team for their contribution to the success of this statistical testing project. M. Houghtaling of the Microcode Tools Group at AdStaR is

recognized for his key insights during the project. Also, we thank the development group, headed by M. Brewer, who engineered the high quality microcode on which this experiment was conducted. Finally, we acknowledge the participation of Software Engineering Technology and the IBM Cleanroom Software Technology Center.

References

1. W. Feller, *An Introduction to Probability Theory and its Application*, Vol. 1, John Wiley and Sons, New York, 1950.

2. H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering", *IEEE Software*, pp. 19-24, September 1987.

3. J. A. Whittaker, *Markov Chain Techniques for Software Testing and Reliability Analysis*, Ph.D. Dissertation, Department of Computer Science, University of Tennessee, Knoxville, 1992.

4. J. A. Whittaker and J. H. Poore, "Markov Analysis of Software Specifications", *ACM Transactions on Software Engineering and Methodology*, Vol. 2, No. 1, pp. 93-106, January 1993.

Appendix A - Formulas for the Analytical Results

The following formulas are necessary to compute the analytical results discussed in this paper.

It is required that all usage chains have a unique invocation state, denoted *Invoke* and a unique termination state, denoted *Terminate* for bookkeeping purposes in the computations. Furthermore, there is a single exit arc from the terminate state to the invocation state with probability one, representing the end of one use and the beginning of the next.

The state transition diagram can be interpreted as a square matrix with the states as indices and the transition probabilities as entries. This matrix is called the *transition matrix*, is denoted P, and has entries p_{ij}. The sum of the entries for each row in this matrix is exactly 1.

The *steady state* probabilities, π, are computed by solving the system of equations $\pi=\pi P$. Each entry π_i is interpreted as the probability with which state i is generated asymptotically in sequences from P.

The steady state probabilities are used to obtain other results as well. The *expected recurrence time* for state i is $1/\pi_i$ and the expected number of occurrences of state i between occurrences of state j is π_i/π_j.

The *expected first passage time* m_{ij}, i.e., the expected number of transitions between states i and j, is computed by solving the system of equations

$$m_{ij} = 1 + \sum_{k \neq j} p_{ik} m_{kj} . \tag{1}$$

The second moment of the first passage is

$$m_{ij}^{(2)} = 2m_{ij} - 1 + \sum_{k \neq j} p_{ik} m_{kj}^{(2)} . \tag{2}$$

The variance can be computed as $m_{ij}^{(2)} - (m_{ij})^2$ with the standard deviation being its non-negative square root.

Additional results are available from the Markov chain by redirecting the arc from *Terminate* to *Invoke* back to *Terminate*. The *Terminate* state is then called an *absorbing* state, and all the other states are said to be *transient*. This reorganization creates a new transition matrix which is denoted P'. The expected number of times state j occurs before absorption is computed from this matrix by solving the system of equations

$$m(j|Inv) = \sum_{k \in \tau} p'_{Inv,k} \, m(j|k) + \begin{cases} 1 & \text{if } j=Inv \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where τ is the set of transient states and *Inv* is the initial state. The second moment (and thus the variance and standard deviation) has a similar form to equation 2 above.

Further analysis is obtained by forcing absorption at some state j (in addition to absorption at *Terminate*) which results in another transition matrix P''. State j is no longer in the set of transient states and both j and *Terminate* are absorbing. The probability that j occurs in a sequence is, therefore, the probability

that absorption occurs at j. This is computed as

$$y_{Inv,j} = p''_{Inv,j} + \sum_{k \in \tau} p''_{Inv,k}\, y_{kj} \tag{4}$$

where *Inv* is the initial state. The expected number of sequences until state j occurs is $1/y_{Inv,j}$ and the expected number of transitions until the arc j,k occurs is $1/(y_{Inv,j} \times p_{jk})$.

The testing chain can also be interpreted as a square matrix with the states as indices and the transition probabilities as entries. This matrix is called the transition matrix and is denoted Q and has entries q_{ij}. The sum of the entries for each row in this matrix is exactly 1.

The reliability is computed by forcing absorption to occur at the terminate state as well as at each failure state. These changes yield a new transition matrix Q'. Any sequence of states beginning at the initial state of Q' will be absorbed in either the terminate state or one of the failure states. The reliability, R, is the probability that absorption occurs at terminate and is computed

$$R_{Inv,Term} = q'_{Inv,Term} + \sum_{j \in \tau} q'_{Inv,j}\, R_{j,Term} \tag{5}$$

where *Inv* denotes invocation of the software, *Term* denotes the terminate state, and τ is the set of transient (non-absorbing) states. Note that $1/R$ is the expected number of sequences of the testing chain until a failure state appears.

The *discrimination* of the testing chain from the usage chain, denoted $D(U,T)$, is computed by

$$D(U,T) = \sum_{i,j} \pi_i\, p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{6}$$

where π is the steady state distribution, p_{ij} is the probability of a transition from i to j in the usage chain, and q_{ij} is the corresponding probability in the testing chain.

The *distance* between U and T_i is

$$d(U,T_i) = \sqrt{\sum_{j,k} (p_{jk} - q_{jk})^2}\ . \tag{7}$$

A MEAN TIME TO FAILURE SOFTWARE TESTING METHODOLOGY

Ian Ferrell, Matt Siegel
ianf@microsoft.com
mattsie@microsoft.com

Desktop Applications Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
(206) 936-8080

KEYWORDS:

MTTF, software evaluation, software reliability, quality metrics, random testing

BIOGRAPHICAL INFORMATION:

Ian Ferrell: Ian received a BS in Physics from Walla Walla College. He worked as a reactor fueling engineer for Ontario Hydro and dabbled in human interface technology at KKG before joining Microsoft. He secretly yearns to someday take a vacation and do nothing but sit on the beach with his loyal, but lonely Siberian Husky...

Matt Siegel: Matt holds a BS from the University of California, Davis, in Computer Science and Mathematics. His specialties are automation and Macintosh software development, although he would much rather spend his time talking about the world's most graceful bird: the penguin.

ABSTRACT:

The thrust of this paper is to report on a practical, real-world implementation of a software reliability measure. We set out with the goal of finding a qualitative measure of software reliability. Toward this, we implement a mean time to failure (MTTF) metric using data from testers and an automated testing system. The tester method estimates the usage level the software can sustain before first failure while the automated system exercises the software according to different user operational profiles. Along the way, we encountered a number of signposts revealing alternate uses for the data and opportunities for future improvement.

WHERE DO WE WANT TO GO?

Our goal is a qualitative measure (or trustable estimate) of any given software release's reliability.

This goal fell out of several concerns that surfaced one drizzly Seattle afternoon; it was one of those retrospective days when work is set aside for a moment in favour of debates over how to test software better and smarter. The concerns were as follows:

- As a product's ship date draws near, it is important to assess each release candidate's "fitness" quickly and accurately as well as have some indication of the product's stabilization trend.
- We need to identify the weakest sections of the software at the most time-critical phases of the project, enabling us to shift focus/resources to affect the greatest change.
- Any method we use must fit within our current framework of test engineering and methodologies. We don't want to change the way people work as much as we want to leverage every effort as much as possible, particularly near the end of the project.
- Along the path to these goals we want to work smart. We need to look for pragmatic methods and data that can be "recycled" to provide the most results for the least effort. We want our methods to be extensible so the experience is easily transferred to other projects.

During this project, we will run across snags and discover workarounds or added benefits to be gained. These "signposts" along our path are valuable insights, often gleaned by hard work or a flash of inspiration. As such, we will pause occasionally throughout the paper to identify any of the signposts we come across.

WHERE WE START:

EXISTING DEVELOPMENT AND TESTING METHODOLOGIES

The project is a commercial software application designed for a broad range of users with differing levels of computer expertise and needs.

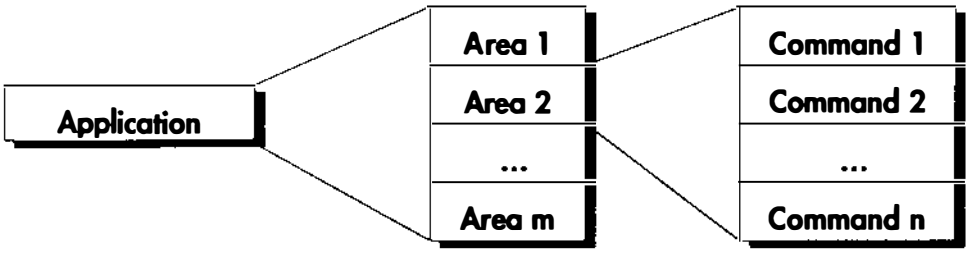
Our group uses the milestone development methodology, where the development cycle is split into several stages, or "milestones", and groups of features/areas are added or modified during each milestone. In general, an area is coded independent of other areas under development during that milestone. At the conclusion of each milestone, areas are completed and tested separately before joining other areas. After the final milestone, all new code and areas are in place, and the product is ready for full-scale feature stabilization through interaction testing.

An area's stability may fluctuate massively throughout the early stages of development as code is added, modified, removed, added, removed, the developer is fired, more code is added, ad nauseam. For this reason, we start our analysis at the code complete point in the development cycle, at the end of the final milestone. We define code complete as the point in the project where each area is considered complete and stable enough to be tested with other areas. All of our calculations and assumptions are generalized for the testing period past code complete, when the majority of testing efforts are focused on the product as a singular application, rather than a disparate collection of independent areas.

AREAS

We have used the concept of an "area" without defining what an area is. The assumption we make above all others is that our product can be broken down into areas. This subdivision covers the entire design, development and testing process where a set of related commands is worked on individually and isolated from other changes in the product. The underlying architecture has been present for several versions of the application, and is a convenient division for data normalization and comparison.

The relationship between areas, commands and the whole application can be thought of as:



Here the finished application is comprised of a number of areas. Within each area are the application commands associated with that area.[1] The entire application fits together like a set of nested boxes.

USER DATA

After releasing our previous version to the public, we created a special "instrumented" build of the application that logged every command used. We distributed this build to a random set of users from our registered user base and collected feature usage data. This data forms a reference point to determine how an "average user" runs the product. The instrumented version recorded each command and how long it took to execute. It also logged the method used to call each command (e.g. keyboard, menu, toolbar). This usage data provides an operational profile for our software product.

From this data and additional marketing research, we identify different groups of users, ranging from the novice/casual user through various incarnations of the "power" user. These breakdowns, together with our knowledge of which features each group tends to use most, present a unique opportunity to evaluate software reliability for different user segments/ operational profiles.

HOW WE ARE GOING TO GET THERE

For our reliability metric, we choose a two-pronged approach. Each prong centers on determining the software's MTTF; one prong is a simplified statistical analysis, the other a targeted–random automated system. The methods are reduced and/or tailored to meet our goals: easy to generate, good fit within the framework of our existing methodology, complement each other's weaknesses, and with a little imagination have many different uses. Each method is described in detail in the following sections.

---

[1] A "command" is the fundamental usage unit in our application. An easy way to visualize a command is to think of an application's macro language. This level of granularity closely approximates the command level we work with. Put another way: think of the standard File/New, File/Open, and File/Save commands as falling under the owner area of File I/O, which is one aspect of the larger application.

AREAS AS NORMALIZERS

The area division is a convenient normalizer for our different data sources. Data from the failure database is based on areas. Our instrumented version gave us data about commands, each of which maps to its owner area. Testing, development and design assignments are also distributed by area. By mapping everything into this consistent set of areas, we normalize data from one software release or data collection method to another.

TESTER/BETA SITE DATA

To measure how testers are using the application, we add a simple command logging mechanism, similar to the instrumented version we sent to our users. This mechanism, however, needs to be much smaller and faster so it will not interfere with the testing process. Consequently, our testing data is not as detailed as the user data; we only collect raw command hits. As each command is executed, a counter for that command is incremented. When an assertion fails[2] or the program terminates, we write the command count array to a disk file. This file initializes the command array when the application is relaunched. Once a new testing release is distributed, we collect and collate the previous release's data files.

The same method gathers data from our beta sites. This provides additional user data for areas that are new or substantially changed in the application. The area usage data also provides a convenient sanity check for our previous version's operational profile; we can notice shifts in area activity and adjust our usage estimations accordingly.

BROAD VIEW OF DATA SETS

In this project, we have our four main data sets: the user, tester, beta, and failure data. The four sets fit together like this:

Data Set	Source	Use
User	Marketing research using the previous public release.	Application operational profiles for different user segments. The profiles define each area's use frequency for a given segment of our user base.
Tester	Testers using prerelease software.	Area hit values (how much an area is tested) used in our MTTF calculation.
Beta	Beta testers using prerelease software.	Highlight shifts in the operational profiles based on new or redesigned features.
Failure	The failure database.	Area failure values (how many times an area fails) used in our MTTF calculation.

OPERATIONAL PROFILES

From the user data and segmentation research, we determine operational profiles for different sets of users. Each set uses the product for different tasks and requires different functionality and feature. For example, beginning users seldom use the customizing capabilities that "power users" use constantly. With the operational profiles, we define typical users within that set and weight our calculations to target specific groups in our broad user spectrum.

---

[2] Asserts, as used in this paper, are a normal method for trapping logic assumptions that fail in code. For example, a developer who assumes a variable passed to a function will always be non-zero can add assert code that fires if the variable is ever zero. Asserts are documented in the ANSI C standard.

STATISTICAL ANALYSIS OF MTTF

MATHEMATICAL MODEL

There are several points to note in our model:

- Our model is a simplification of standard MTTF models. The implementation is mandated by what data we can collect, and how that data can reflect the software stability. As such, and in keeping with our stated goal to find a qualitative measure of software stability, we ignore the raw numerical results and look instead at the trend. We are not seeking a particular magic number, but rather some indication of what direction the software's stability is moving and how fast it is getting there.[3]
- Areas play a major role in our calculations, forming the basis upon which the model is founded. Our assumptions for areas, outlined earlier in the paper, extend to the model, so while we acknowledge individual commands within an area may involve portions of code from each other, they are a singular, independent unit outside of the area (much like a black box). We assume each area is exclusive of all other areas; Area X will not contain pieces of Area Y or vice versa.
- There are differing opinions on how to map from test data to calendar time. Rather than track CPU usage, (which varies greatly across the possible range of PC hardware), or calendar time (which varies from user to user), we chose to only track area hits.[4] A hit corresponds to one command action, as described in the previous section.
- Failure is defined as any non-duplicate program failure logged on the failure database. These may include program crashes, functional failures, feature design issues/requests, interface problems, display glitches and international needs.

Our method determines the mean number of hits each area can sustain before failure. From these numbers, we pick the limiting area (which area will fail first) based on a particular operational profile. Once we identify the limiting area, we determine the mean number of hits before the product fails.

To determine the mean number of hits each area can take before failing, we calculate the hazard rate for the area using data from each tester's machine and our failure database:

$$\theta_i = \frac{\#\ of\ failures\ in\ area\ i}{\#\ of\ hits\ in\ area\ i} \tag{1}$$

This hazard rate is the likelihood of a command in area i failing. It is *not* the time to failure. The mean number of hits for area i, (h_i), before failure is:

$$h_i = \frac{1}{\theta_i} \tag{2}$$

[3] We definitely want a magic number in the future; this project is a first cut at determining a MTTF. As our data experience grows, we can draw more meaning from the raw values and have a basis for comparison.

[4] An experienced user on the fastest hardware money can buy may hit a defect orders of magnitude faster than a novice on a slow machine, even though they take an identical path.

At this point we must determine which area is the limiting factor: which area will fail first? We could take the "raw" first failure area (the one with the lowest number of hits before failure), but that does not reflect how people use the product. If the first failure area happens to be some obscure area that is rarely used, the product may appear less stable than it really is. We need to reflect real-world product usage in our calculations.

Weighting each area's failure point by its use frequency pushes high-use areas to the front of the line, while restraining secluded areas that are not commonly used. The operational profile determines which area will fail first; that area is the limiting area for the user profile. Toward this end, we normalize the h_i's:

$$\overline{h_i} = \frac{h_i}{\sum h_i} \tag{3}$$

Let u_i be the probability of area i's usage (from the operational profile). To determine the limiting area x:

$$x = i \text{ to minimize } \frac{\overline{h_i}}{u_i} \tag{4}$$

We then compute the total number of hits, (H), the program as a whole can take:

$$H = \frac{h_x}{u_x} \tag{5}$$

Appendix I is a demonstration of the MTTF calculation using a sample application.

PROBLEMS WITH THIS MODEL

- **New Features**

  We rely on data from our previous version. This data does not take into account new features and major changes to existing ones. To solve this problem, the data from our beta sites is stack ranked (by usage) with the previous version's data. We then merge the data for the new areas between the previous version's data. With this method, we get a reasonable utility range for each area rather than a more subjective measure of assumed utilization.

- **Failure database queries**

  One difficulty with the failure database is the degree of overlap between feature areas. Each failure in the database is classified according to the area where it is found. Unfortunately, the area designations in the database do not match perfectly with the areas we are using; they contain relics of terminology or design that changed over the project's history. Once we add an area term to the list of possible database entries, it cannot be deleted from the list. Since testers classify a bug's area when they create a new bug description, this leads to a great deal of creative bug area distribution. On top of this, there is also a "sub-area" field to further complicate matters. In most cases, the areas are reasonably easy to determine; for a few situations however, the special-case handling was quite complex.

- **Mean Time To Failure**

  Our ideal goal is a mean *time* to failure, however, implementing this given our available tools and data requires either loose mathematics or a substantial investment in changing the data collection method. Yet, even with our rudimentary scheme, we can calculate the mean *hits* (command calls) to failure. This measure analyzed over a set of testing releases yields an indication of the software's stabilization trend.

MTTF SIGNPOSTS

Each piece of our solution was designed to accomplish its goals as completely as possible within the given constraints. In addition, great effort was put into seeking alternate means of analyzing the data. Each of these alternate methods was worth the effort of collecting the data, but when combined with each other and the overall analysis, the group constitutes a huge return on investment.

- **Hazard rate vs. "buggy" area**

  Prior to this analysis, any determination of "buggy" product areas was subjective. Raw failure find rates from the database depended on how strenuously the tester tested the area.[5] Lacking in this analysis is a weighted measure of area (in)stability taking into account the level of testing effort. The hazard rate, with its inherent dependence on testing effort, develops a clearer image of where the unstable areas lie within the product. This analysis is invaluable to testers assessing an area's risk and/or managers looking for areas to reallocate.

- **Critical areas**

  One of the steps in determining our MTTF is finding $Area_x$, the critical area. This measure takes into account the area's hazard rate, combined with the area's likelihood of use. But once $Area_x$'s hazard rate lowers, another area becomes "critical". We can look at each area and determine the areas around the critical stages. Once we know areas that are critical, we can focus the test team's attention on those areas where it will do the most good.

- **Area usage and area risk**

  By collecting usage data from testers and from real users, we can normalize and compare the two, noting discrepancies. Testers often test software different from the way users use the software because the design has changed in the new version or a particular testing release is unstable in some areas. A slight variance between tester and user usage is not necessarily bad or indicative of poor testing, but any major difference should be examined. Usage data also shows which areas are rarely used; this information is invaluable to the marketing and design groups.

- **Different users see the product differently**

  Due to the reliance on frequency data for each area's use, the calculations can be adjusted to approximate different user profiles. For example, our earlier example in which the beginning user rarely uses application customization commands, but an advanced user may use them heavily. We determine the different operational profiles with an instrumented version and then plug them into the calculations. This flexibility allows us to ascertain critical areas for different sets of users and assess risk.

---

[5] A low failure find rate means that the area could be either stable or poorly tested.

AUTOMATED TESTING

The statistical model of MTTF use is useful in its predictive nature. However, there are some simplifications in the model that cause it to inadequately describe the software engineering and testing process. For example, some areas are gateways to other areas; if one of these is unusable, all areas beyond the green door are also unusable.[6] Our statistical method does not take interactions between areas into account.

In addition, there is the constant problem of test automation. While the theory of a testing machine is quite seductive, too often any implementation proves to be quite expensive for the results gained. True random test automation produces nothing but truly random results, and pre-manufactured testing scripts fall prey to Beizer's *Pesticide Paradox* where they grow ineffectual at finding any bugs.

To address these issues, we design and implement an operation profile-based, random automated testing system, the automated test Weasel.[7]

The Weasel is driven by our user operational profiles and executes area-based pre-programmed commands in a random order with random parameters based on typical user input. Since the randomizer is based on the operational profile, Weasel acts very much like the user segment the operational profile is set up for.

WHY OVERLAPPING METHODS?

The two methods (automation and statistical) of determining MTTF have different base assumptions. The statistical method relies on many numeric approximations while the Weasel will never be as good at testing as a human since its failure definition is too limited. With these two methods used in parallel, we hope to come up with similar MTTF trends.

Comparing the correlation between our automation and statistical results will indicate how closely our automated Weasel can approximate standard testing efforts. While the Weasel can never replace human testers, a high degree of correlation between the automated system and our testing data will verify our methods and validate the assumption that a high level of stability among individual areas translates to a high level of stability across the entire product. This will also justify our argument that a random-weighted automation method most closely approximates user behaviour.

IMPLEMENTATION

Weasel takes the user operational profile and generates scripts based each area's frequency of use. Each script is then run inside of a test driver that controls which random user action is performed.

The Weasel logs its progress through the tests in an executable form, creating a reliable method of reproducing any problems and regressing each failure automatically after it is fixed. It also allows us to keep a precise record of all tests that have been run as a regression suite for subsequent testing releases.

Weasel is designed so portions of the log may be run independently to minimize the number of steps needed to reproduce failures. By adding new functionality beyond the random

---

[6] For example, the boot/initialization code, while independent of other areas, can seriously block testing if it fails.

[7] The name fell out of our desire to have a testing monkey that was cunning and devious in its testing methods, not a rote automaton plugging through predefined scripts. After having several ferrets around the house as pets, "Weasel" seemed to be an apt moniker...

number range of existing functions, the system is expandable without perturbing existing scripts.

The basic method of fault detection Weasel uses is in-code assertion failures, document corruption, and general protection faults. We chose to only look at these failures for several reasons. Each feature area has differing functionality, and a host of potential items to verify, resulting in a disparity between areas in action quantity and verification. Also, these methods are what we categorize as "severity 1" bugs, which maps directly to our failure database. All other bug severities are substantially more subjective.[8]

The Weasel runs through a script until it hits a critical error that causes it to stop. Once this occurs, we analyze the command hit record and post failures on our database. Weasel is run many times, and the average of these hits to failure is determined.

ANALYSIS

The area hit totals we collect from the automated system are tracked for each testing release, along with the MTTF results from the statistical analysis. We compare the numbers between methods and with the previous release to see trends. Since this is our first attempt at gathering and analyzing automated MTTF data, this particular facet does not have extensive goals beyond successfully running and corroborating our statistical data.

WEASEL SIGNPOSTS

This initial effort at an autonomous automated system, and has revealed much about the considerable weaknesses and expense of most automated testing architectures.

- **Weasel vs. "Random Testing"**
  The Weasel's randomness is not true randomness; rather, it randomly runs set micro-scripts with random parameters driving the scripts. This is different from a "monkey" that hits random keystrokes at random places. Weasel's reliance on semi-randomness more closely mimics real world users: a user is far more likely to select random pieces of a document and act on them than he is to start typing in a dialog.
- **Pre-processed bugs**
  The end of each Weasel run is caused by a failure. Since we log scripts as we go, each failure is completely reproducible, simple to narrow down since we have a clear list of what steps led up to the failure, and is already automated. After the failure is posted, we can add this to our existing automated system to ensure no regression of the fix occurs.
- **Simple area targeting**
  Since the Weasel is based on the operational profile, we can easily increase the pressure on a specific area by changing that area's profile term in the spreadsheet. This characteristic allows us to shift Weasel's focus from emulating an average user to emulating a user segment that stresses a specific functional area. If a tester is concerned about the widespread ramifications of a particular code change, he could simply increase the stress level on areas he feels are most affected and fire away.
- **We realize the Weasel is not the Holy Grail**
  There is an easy tendency to place the Weasel high on a pedestal and urge the elimination of all testers in favor of automated Weasels or some automated system. No,

[8] Considering only blatant errors without considering functionality errors is a weakness of this system. Unfortunately, automating verification routines outside of catastrophic failures is beyond the scope of this project. We hope to remedy this in the future.

no, no! Weasel is only a tool, a reasonably cheap tool, that allows us to automate testing in a way that we could not before. Tester-written automation scripts will remain static snapshots of a test case, the Weasel can use the product like users, within admitted limitations. This is a point we cannot stress enough. Whatever methods we can use to cheaply move the Weasel toward mimicking a real user will increase the tool's usefulness, but the smartest weasel in the world cannot recite poetry or do change testing.

CONCLUDING SIGNPOSTS

OPERATIONAL PROFILE/STATISTICAL SIGNPOSTS

- The operational profiles highlight area use across the application. This data identifies commonly used areas as well as the not-so-common areas. Some areas are used by such a small number of people that it is more likely someone will involuntarily activate the area than deliberately use it. We found quite a few of these gems hidden in our user profile data, and they serve to illustrate the need for good design and implementation.
- The operational profile enables us to assess the application's stability with greater accuracy than raw data analysis. With the profile, we see the product as our users do. Often, this reveals new test scenarios and interactions between areas.
- Dividing the product into areas greatly simplifies data analysis. Data is transferable across disparate application versions and collection methods. Project statistics expressed with this common reference are easier for internal and external groups to understand and compare.
- Hazard rates are ideal to identify "risky" areas; areas that may need more testing or have particularly buggy code. Analyzing hazard rates throughout the product cycle reveals these areas early enough for corrective action.
- Research of this type offers an ideal opportunity to get the testing, development, design and marketing departments working and communicating better together. The additional benefits we outline throughout the paper can work for all aspects of the organization; there's something in here for everyone. This may seem to be a pie-in-the-sky "feel-good" benefit, however, it is a valid concern, particularly in larger product groups that have little interdepartmental communication.
- Collecting operational profiles from the beta sites not only reveals shifts in application use but complements usability research as well. An early perspective of the product's operation can positively affect every group involved, from design and documentation through development and testing.

DATA COLLECTION SIGNPOSTS

- Automating data collection and manipulation pays rich dividends down the road when you must balance testing responsibilities with a mound of data needing analysis. To date, we have managed to automate about 90% of this process. While the automation is not mind-numbingly fast, it does allow us to get our other work done at the same time.
- Recording use data requires a different process for each application. Internal program architectures are distinct and vary greatly depending on the product's purpose. All applications generally have some method of distributing user requests/actions. The recording method should tie into the application at this point.

- It is important to select a good recording method early in the project to keep data consistent. The procedure should not adversely affect application performance or hinder normal testing efforts. Optimization is the key: the code will run every time a user or tester performs some action and they should not be able to detect any difference in application behaviour.
- Identify what data you need and structure the recording scheme toward this goal. Eliminate over-ambitious data-recording methods that archive every possible aspect of a command. At the same time, remain alert for any signposts pointing to data that could serve double-duty; a data point that is worthless on its own could be invaluable when examined next to other data points you have to collect.
- Any data recorded must be collected. Keep the data files small, and look for schemes to automate collection. Toward the end of the project, testers have many things to consider; the last thing they worry about is copying your data file to a server. Electronic mail or network distribution methods that operate automatically are the best way to go.

WEASEL SIGNPOSTS

- The "plug-in" operational profiles allow quick changes to Weasel's testing personality. Altering the profile enables us to stress specific areas or analyze the product's stability for different user segments.
- The Weasel detects failure classes human testers have difficulty detecting. For example, the "long use" scenario is often impractical for human testers to duplicate due to work hour limitations and the complex nature of long use sessions. Weasel is content to work for multiple hours on the same document and leave a precise log file of all actions performed.
- All Weasel log files are executable. Once a failure is detected, the log file is manually reduced to the minimal steps needed to reproduce the failure and added to the standard acceptance test suite.
- Weasel is random-weighted automation method. It neither falls prey to Beizer's *Pesticide Paradox*, nor does it attempt to write *Hamlet* one character at a time. The operational profile guides Weasel to emulate a user, while the random element mimics real-world scenarios. We believe the Weasel represents the future of automated testing.

APPENDIX I: A SAMPLE MTTF CALCULATION

For this example, we will use a simple application that has five functional areas. (All of the data is simplified for demonstration purposes.) Depending on the application, mapping commands to areas may or may not apply, so we have left this exercise to the reader. Our illustration begins with the assumption that the fictional application and sample data has already been divided into areas.

We start with our three main sets of data: the user, tester and failure data sets. Notice that raw analysis of the bug find rate suggests that Area 3 is the "buggiest" area:

	Area 1	Area 2	Area 3	Area 4	Area 5
User Data (from operational profile, u_i)	50%	20%	10%	10%	10%
Area Hits (from tester data)	700	800	750	200	1000
Area Failures (from failure database)	7	16	25	10	20

EQUATION 1: DETERMINING HAZARD RATE

From this data, we determine the hazard rate by dividing the failures in a given area by the number of hits on the area:

	Area 1	Area 2	Area 3	Area 4	Area 5
User Data (from operational profile, u_i)	50%	20%	10%	10%	10%
Area Hits (from tester data)	700	800	750	200	1000
Area Failures (from failure database)	7	16	25	10	20
Hazard Rate (θ_i)	0.01	0.02	0.03	0.05	0.02

EQUATION 2: DETERMINING MEAN HITS TO FAILURE

Inverting the hazard rate yields the mean number of hits until failure:

	Area 1	Area 2	Area 3	Area 4	Area 5
User Data (from operational profile, u_i)	50%	20%	10%	10%	10%
Area Hits (from tester data)	700	800	750	200	1000
Area Failures (from failure database)	7	16	25	10	20
Hazard Rate (θ_i)	0.01	0.02	0.03	0.05	0.02
Mean Hits to Failure (h_i)	100	50	30	20	50

EQUATION 3: NORMALIZE THE MEAN HITS TO FAILURE

We normalize the mean hits to failure to determine the limiting area. If we wanted the "raw" limiting area(s), it would be the lowest value(s) at this stage. For our sample calculations, the raw limiting area is Area 4:

	Area 1	Area 2	Area 3	Area 4	Area 5
User Data (from operational profile, u_i)	50%	20%	10%	10%	10%
Area Hits (from tester data)	700	800	750	200	1000
Area Failures (from failure database)	7	16	25	10	20
Hazard Rate (θ_i)	0.01	0.02	0.03	0.05	0.02
Mean Hits to Failure (h_i)	100	50	30	20	50
Mean Hits to Failure (Normalized)	0.4	0.2	0.12	0.08	0.2

EQUATION 4: DETERMINING THE LIMITING AREA

To determine the limiting area, we divide the normalized mean hits to failure by the user percentage from our operational profile. This weights the hits according to the area's use frequency. The lowest value(s) at this stage are in the limiting areas:

	Area 1	Area 2	Area 3	Area 4	Area 5
User Data (from operational profile, u_i)	50%	20%	10%	10%	10%
Area Hits (from tester data)	700	800	750	200	1000
Area Failures (from failure database)	7	16	25	10	20
Hazard Rate (θ_i)	0.01	0.02	0.03	0.05	0.02
Mean Hits to Failure (h_i)	100	50	30	20	50
Mean Hits to Failure (Normalized)	0.4	0.2	0.12	0.08	0.2
Mean Hits to Failure (Over user data)	0.8	1	1.2	0.8	2

So, for our sample application, the limiting areas (first to fail according to the operational profile) are Area 1 and Area 4. Where the raw limiting area flagged Area 4, using the operational profile flags Area 1 as well. This signifies that the profile's average user is as likely to run into his first failure in Area 1 as in Area 4.

We compute the total number of hits the program can sustain before failure by dividing the limiting area's mean hits to failure by the operational profile (usage) percent. Using the data from Area 1 as a sample (either limiting area will give the same results):

$$\therefore \quad H = \frac{h_x}{u_x} = \frac{100}{50\%} = 200 \; hits$$

REFERENCES

The following list of references outlines research reports and publications we have consulted during our research and the preparation of this paper.

- Alexander, C., S. Ishikawa and M. Silverstein. 1977. *A Pattern Language.* New York: Oxford University Press.
- Beizer, B. 1990. *Software Testing Techniques.* New York:Van Nostrand Reinhold.
- Dixon, F.W. 1944. *The Melted Coins.* New York:Grosset and Dunlap.
- Duran, J.W. and S.C. Ntafos. 1984. An Evaluation of Random Testing. *IEEE Transactions on Reliability.* SE-10:438-444.
- Grady, R.B. 1992. *Practical Software Metrics for Project Management and Process Improvement.* Englewood Cliffs, NJ:Prentice-Hall, Inc.
- Hamlet, R. and J.M. Voas. 1993. Software Reliability and Software Testability. Presented at *Quality Week.*
- Hoshizaki, D.O. 1993. Risk-Based Refinements for the ANSI/IEEE Test Design Specification. Presented at *Quality Week.*
- Masuda, Y., N. Miyawaki, U. Sumita and S. Yokoyama. 1989. A Statistical Approach for Determining Release Time of Software System with Modular Structure. *IEEE Transactions on Reliability.* 38:365-372.
- Musa, J.D. 1989. Tools for Measuring Software Reliability. *IEEE Spectrum.* 26:39-42.
- Musa, J.D., A. Iannino and K. Okumoto. 1987. *Software Reliability: Measurement Prediction, Application.* New York:McGraw-Hill Book Company.
- Negroponte, N. 1970. *The Architecture Machine.* Cambridge:MIT Press.
- Pfleeger, S.L. 1992. Measuring Software Reliability. *IEEE Spectrum.* 29:56-60.
- Tufte, E.R. 1990. *Envisioning Information.* Cheshire, Connecticut:Graphics Press.
- Voas, J.M. and K.W. Miller. 1993. Semantic Metrics for Software Testability. *The IEEE Journal of System Software.* 20:207-216.
- Voas, J.M., J.E. Payne and K.W. Miller. 1993. Designing Programs that are Less Likely to Hide Faults. *The IEEE Journal of Systems Software.* 20:93-100.
- Walsh, J. 1993. Determining Software Quality. *Computer Language.* 10:57-65.

Beyond Styleguides: Implementing User Interface Consistency in a Distributed Organization

F. Beachley Main

Halliburton Energy Services
11255 Kirkland Way
Kirkland, WA 98033
Phone: (206) 822-5200
Email: beachley@sgihbtn.com

Abstract:
Developing applications with a consistent user interface has many benefits to both the end user and the developing organization. Most organizations approach the problem by specifying and then developing a style guide, this is not sufficient. Consistency is inherently a relational concept, hence communication among the application teams is as much a part of solving the problem as is resolving particular technical issues.

A matrix team consisting of members from both product and functional groups is required. The goals and responsibilities of the team must be clearly defined and accepted. Many types of deliverables are useful in developing consistent user interfaces. The process must have a committed executive sponsor to be effective. While there are several dangers in the process, the results do justify the investment. The process needs to become an integral part of how the organization develops software.

Keywords and Phrases:
user interface quality, user interface consistency, style guides, user interface consistency process.

Biographical sketch of author:
Beachley Main holds an M.S. in geophysics from the University of Houston. Prior to working for Sierra, he worked for Shell Development Company creating a variety of workstation-based scientific visualization applications. He is currently a research manager for Sierra Geophysics. The group that he manages is responsible for developing and implementing a consistent user interface style throughout the Halliburton Energy Services Group Companies. The group is also responsible for implementing user interface components that promote this style as well as improve application programmer productivity. Over the past two years through the user interface work group process, Halliburton has made solid advances in both user interface consistency and usability. This has made a strong marketing impact for the company at recent trade shows.

Introduction

Consistency is one of the most desirable aspects of user interface quality. It is a very important technique towards the primary goal of developing applications that are highly usable. So why is it so rarely achieved ? Because implementing user interface consistency in an organization needs to be an ongoing process. A static style guide will not solve the problem.

This paper describes, from a management perspective, the advantages of developing user interface consistency, and two approaches to solving the problem: through the use of style guides and through the development of an organizational process. This proven process is described in detail. Lastly, some conclusions are provided along with warnings on the dangers of implementing the process in your organization.

The material presented here is the result of experience gained working to implement consistent user interfaces for a family of applications being developed at six locations in three states and two countries. The family of applications represents roughly twenty-five development projects over the course of two years. Additionally, this paper is the result of research and discussion with others who are working to implement consistent user interfaces for their software products.

Definition of Consistency

What do we mean by "consistent user interfaces"? The definition of consistency in user interfaces is a bit like the definition of quality that states "I can't tell you what it is, but I know it when I see it". Consistency defines a relationship between two or more objects that show a uniformity, compatibility, or agreement of parts or of the whole. It does not mean that they are the same. Consistency in user interfaces means that when a user sees a particular symbol or component on the screen, that the semantics that the user attributes to that component do not vary from one use of the component to another (common look). Consistency also means that when the user interacts with the component, that for each action the user performs on the component, the component responds every time in a manner the user perceives as the same (common feel).

Consistency also has some dimensions that need to be considered so that the organization can understand what it is trying to accomplish. Consistency can mean consistent over time with previous versions of a product. It can mean self consistency within a particular product. It can mean consistency of the interface of a particular product across multiple hardware and software platforms or it can mean that the product is consistent with un-related applications on the same software platform. Consistency can also mean that the software application performs in a manner that a user of a non-automated system performing the same type of task will recognize that task in the software system.

The emphasis here is on how to develop applications that are consistent with other applications developed by a particular organization and targeted at a single software platform (e.g. UNIX

and OSF Motif). The dimension of consistency with previous versions was subordinated to the requirement to be consistent to the target platform. Consistency to the non-automated task performance was also considered quite important.

Advantages of Consistency

The user advantages of a consistent user interfaces include ease of learning, ease of use, improved satisfaction, and less resistance to using new software. The advantages to the user's company are lower user training costs, reduced requirements for user support, and lower training costs for the support personnel.

The advantages to a company selling software are even more dramatic. The company will be able to evolve its products in a controlled manner. Once particular design issues are identified, the new solution can be put in place in new product versions simultaneously. The organization developing software will be able to reduce its design costs; many aspects of the design are already defined by the created standard. Other benefits to a vendor company:

- Reduced implementation cost through reuse of code that implements the standard. If arbitrary design decisions can be standardized then developers can use one implementation.
- Reduced maintenance costs through systems built on previously tested designs and code.
- Ease of use translates into increased consumption and sales. When the users cost of learning a new system is reduced, they will be able to absorb new applications.
- Better product definition through family resemblance. Buyers will be more likely to buy follow-on products that look and feel like the ones they have already learned.
- Improved usability through a standard based on research. Once you have eliminated the tendency to have multiple arbitrary solutions to the same design problem, you can afford to spend more effort to create a single superior solution.
- Designers can focus on the unique aspects of their product's user interface. They do not need to spend as much effort on the parts of the application that are the same as the rest of the product family.

It is clearly in the best interest of any organization that develops software to work towards developing consistent user interfaces across its product line. While there are significant benefits to the end user, the benefits to the software developer are even more striking.

Style Guide Approach to Consistency

The first step an organization typically takes to develop user interface consistency is to specify the use of an existing style guide for the GUI they are using (e.g., OSF Motif Style Guide). Such style guides are intentionally high level and general in the guidelines that they specify so that they are equally applicable to any application domain. This is appropriate so that they do not preclude solutions that are highly usable for specific application domains. Developers adhering to these style guides and their managers quickly come to understand that while necessary they are not sufficient. The user interfaces of the resulting applications are only superfi-

cially consistent at best.

The next step the organization is likely to take is to develop a custom style guide to extend the general guidelines to apply to their business domain. This process uses a lot time from the organizations best user interface developers. This custom style guide is likely to go through several revisions becoming increasingly thick (and expensive). The expectations for this effort will not be achievable. To the extent that the project is successful in specifying enough detail to constrain developers, it will create a document that is so large that developers will not read it. The OSF Motif Style Guide is a good example of the natural evolution in size of a style guide. The original Revision 1.0 document contained 130 pages. With revisions 1.1 and 1.2, the style guide continued to grow. Finally, the draft version for Revision 2.0 was released in the spring of 1993 with 631 pages.

Surveys show that the vast majority of developers who use a style guide do not read it from cover to cover. They typically scan it to understand the general context, and then use the table of contents and index to find answers to specific questions. If they do not consider, at the time that they are making a design decision, that there might be a relevant guideline or if the guideline is too difficult to find in the style guide, then the design decision will not take advantage of the considerable effort that went into developing the guideline.

If there is not a process in place in the organization to develop style issue resolutions, communicate the style guidelines, communicate the importance of adhering to the guidelines, and to review and verify adherence to the guidelines, the user interfaces developed by the organization will not be consistent.

The Process Approach

The effective approach for the organization that is committed to developing user interface consistency for software either for internal use or external sale is to establish and maintain an ongoing group-oriented process. The key elements of the process are:

- Make user interface consistency the primary responsibility of a functioal group. They must know their mission and believe in it.
- Make user interface consistency a responsibility of a member of each product development team.
- Employ every means at your disposal to keep members of the functional group and representatives of the product teams sharing and exchanging information. Communication is the glue that holds the process together.
- Define appropriate deliverables that will effectively communicate both the design decisions and the rationale for the decisions.
- Ensure that management above the product manager is committed to the effort and support it as a first priority.

The following sections will discuss each of the elements in some detail.

Responsibilities and Composition of the User Interface Team

The user interface team is a functional group whose primary responsibility is to facilitate the definition and development of the company's user interface style. This does not mean that they are expected to create the style as much as it means that they are responsible for ensuring that it is developed. For the style to be accepted by the organization's developer community, the style must address the particular needs of the company's problem domain and it must represent the combined expertise of the company's designers.

The team should consist of a strong lead with a background in human computer interaction as well as experience with the particular GUI the organization is using. The team should have from one to several software engineers to develop prototypes for testing design proposals. The engineers will also develop example and demonstration programs to communicate the company's style. The team will also need a graphic design artist to develop the appearance of individual interface components. The team will also need at least a part time technical writer and part time usability testing professional.

Some of the responsibilities of the team are:

- Facilitate meetings of the User Interface Work Group
- Document and distribute meeting minutes
- Document and distribute design issues and UI Models
- Moderate a company-wide news group
- Keep the news group active by asking questions, answering questions, and posting articles
- Conduct technical reviews
- Conduct verification and validation reviews
- Develop proposals for solving design issues
- Develop prototypes of design proposals and usability test them

Responsibilities and Composition of the User Interface Work Group (UIWG)

The User Interface Work Group should consist of a member from each software product team or product group that is to use the organization's style. The members will be the persons responsible for the design of the user interfaces of their respective products. The work group should also have the members of the user interface team as part of all the group meetings.

The responsibilities of the UIWG members include:

- Promptly reviewing submitted materials. Not only will the member who submitted materials learn from the comments, but other designers will learn from the discussion. Exposure to interfaces of the other products will also promote consistency in even small and subtle details.
- Submiting project user interface artifacts for review early and often throughout the project. These items can be early design documents, mock-ups of the user interface, prototypes, and eventually the final product.
- Attending UIWG meetings and sponsored classes. Common experience gained from face to face discussions is the strongest reinforcement to design decisions created through consensus. Likewise, a common training background promotes understanding.

- Submitting and responding to news group questions.
- Completeing agreed upon actions in a timely manner. The representative is actually matrix managed into a UIWG self-directed team and must have the time to complete requested tasks.
- Adhering to UIWG resolutions.

Communication is a Critical Element

Communication is a critically important aspect to making the process of developing consistent user interfaces work. The only aspect of the process that is more important is a shared understanding and commitment to the goal of user interface consistency. The UIWG must function as a team across all project teams. The only way that this can work is if the members of the UIWG team have enough exposure to each other and to the products that the other members of the team are developing. This communication is made more difficult when the team members are at different locations. Even being on different floors in the same building or different buildings in the same complex can decrease the amount of contact that results in informal "hall meetings".

There are many mechanisms for promoting communication among the team members:

- Group meetings are the most expensive, but also the most productive means of communication. Meetings have the greatest ability to enhance the relationship rather than the task aspect of the team. People tend to be more cordial and understanding in a face to face situation.
- Teleconferencing is less expensive than airfare, yet still has some of the advantages of a live meeting However, the time delays introduced by the hardware and adjustments of the hardware are distracting and hinder open discussion.
- Phone conferences still retain the subtleties of the voice while losing the body language. The emotion in the voices carries critical information than electronic mail and is more interactive than electronic mail.
- Electronic mail accentuates the task rather than the relationship. Communication in this manner is very good for preparing and evolving carefully worded opinions; keeping a discussion going in the midst of other activities. The lack of direct contact can lead to "flaming" the opinions of others (revert back to the phone to heal hurt feelings).
- News groups make a good repository for summaries of electronic mail discussions and group documents, but are not interactive.
- A shared computer account is an excellent means for storing group documents and prototype programs for review. The ability to run early prototypes and current releases of each other's programs is essential. Only by working with each other's programs can the group develop a shared vision for the companies user interface style.

Appropriate Deliverables to Communicate Consistent Style

Some of the deliverables that are produced by the user interface team and associated developments teams are:

- Instead of (or in addition to) writing a style guide, develop a set of user interface issues tracked using the Issue Based Information System (IBIS) notation. IBIS is a notation to collect and structure decision rationale. It is a refinement over meeting minutes that

reflects the structure of the natural decision making process. The elements of the notation are the issue, proposals, arguments for and against a particular proposal, and lastly a decision to accept one of the proposals or a specific action needed to gain more information to resolve the issue. The most significant aspect of the notation is that it captures the rationale for the decision which removes the need to revisit the discussion and serves to train new members of the group.

- Styleguides and Issue tracking while important can not possibly provide enough detail that designers would be able to implement larger components identically. For example, if two designers were independently developing a dialog to provide printing capabilities, the style guidelines would not be able to specify precisely where on the dialog the page orientation or margin controls should appear. User interface models are a solution to this problem. When a commonly used dialog is identified by the UIWG, a UI model is developed for it using the control-action-response (CARS) notation and a drawing.
- A toolkit that implements components described by the style guidelines and UI models is the most efficient and effective means of producing applications that are mutually consistent. The efficiencies of reuse of design that UI models represent is increased when the actual code is reused through the toolkit. The use of the toolkit also improves the efficiency of the validation reviews as many of the parts have already been reviewed separately. It is effective in that developers using the toolkit must go out of their way to deviate from the agreed upon style.
- Exemplary applications that instantiate the common UI present give the developers a clear picture of what the guidelines intend. They are less subject to errors in interpretation than are written guidelines.

Management Commitment and Support

Corporate management must be truly committed to user interface consistency of its software products for the process to work. Products must not be delivered until any variances found in the final user interface review are fixed or justified on grounds of usability or user needs. Product managers must be made to understand that this is a priority over the temptation to add that one last feature that may only be used by two percent of the customers.

The needed resources must be allocated to the process. The schedules of the individual product projects must plan for the time that their representative will be pursuing activities related to his/her participation in the UIWG. Travel budgets must be made available for the work group meetings. When the process has been successful on the first round of products, the effort must be maintained for the next round as well.

Lastly, senior management must make sure that the organization capitalizes on the gains that it has achieved through implementing user interface consistency. Promote this aspect of your product quality to the press. Make sure that your marketing and sales staff understand the benefits of user interface consistency and that they communicate this to your customers.

Dangers of Consistency

The most important goal for implementing user interface consistency is to produce software applications that are highly usable. By usable, I mean that the applications have the right mix of several factors: time to learn, retention over time, speed of performance, rate of errors, and subjective satisfaction. Consistency is a *means* of achieving this goal, *it is not the goal.* It is only professional judgement supplemented with rigorous usability testing that will enable designers to decide when inconsistency will yield a more usable product.

There is a danger that the style will not be evolved. To keep products fresh and competitive, new methods of human computer interaction must be researched and innovative technologies built into new versions of the style. These innovations need not be based on fundamental research. Depending on the market, it may be sufficient to closely track related and competitive products for their innovation.

Conclusion

The results of our experience implementing user interface consistency for a suite of applications indicates that the results do justify the costs and the risks. The key factors in developing a successful process are senior management commitment to the process, a core team of designers and software engineers from both the product teams and the functional group that adopt making usable and consistent user interfaces as their personal mission, identifying and producing the appropriate deliverables, and lastly establishing and maintaining the various channels of communication for the team. The process will not work as a quick fix; it must be an ongoing integral part of the way your organization develops software if it is to be successful.

References

Katzenbach, M. and Smith, D. (1993). T*he Wisdom of Teams: Creating the High-Performance Organization.* Boston, MA: Harvard Business School Press.

Nielsen, J. (1989). *Coordinating User Interfaces for Consistency.* San Diego, CA: Academic Press.

Open Software Foundation (1990). *OSF/Motif Style Guide.* Englewood Cliffs, NJ: Prentice-Hall.

Shneiderman, B. (1987). *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addision-Wesely, MA.

A Case of Establishing a Coding Standard

Spass Stoiantschewsky

James Teisher

Abstract

Identifying shortcomings in the software development process, the root causes, and the lessons learned in the process of implementing the corrective action, using a real-life case of the establishment of a software coding standard.

Biographies

Spass Stoiantschewsky is a software engineer with Credence Systems Corporation in Beaverton, Oregon. He is a member of Credence's Standards Committee as well as the Metrics Committee. He received a BSEE at Carnegie-Mellon University in 1984, a MEng. in Computer Systems Engineering at Rensselaer Polytechnic Institute in 1987, and a MBA at Portland State University in 1992. He previously worked at Tektronix since 1987, prior to the acquisition by Credence in 1990.

James Teisher is a senior software engineer with Credence Systems Corporation in Beaverton, Oregon. He is chairman of Credence's Standards Committee as well as a member of the Tools Committee. He received a BS in Computer Engineering from the University of New Mexico in 1982. He previously worked at Tektronix since 1982, prior to the acquisition by Credence in 1990.

I. Introduction

Credence Systems Corporation is a supplier of semiconductor test equipment. More than 300 employees work at three primary sites (Beaverton, OR, Billerica, Mass. and Fremont, CA) and at field offices around the world.

Over the last three years, Credence has followed a technology acquisition strategy that resulted in a series of planned mergers and acquisitions in order to build engineering and manufacturing capability to provide a wider range of semiconductor test equipment and related services. In the process, Credence has had to learn to manage activities within a "melting-pot" of various engineering and manufacturing approaches. Unification of these approaches, such as those relating to software engineering are a major management goal.

In 1991 Credence began a drive towards quality in all aspects of the corporation. In this ever increasingly rigorous quality program, everything focuses on the concept that "Quality" is defined as conformance to requirements. The company offers mandatory corporate wide training programs for all employees to ensure that everyone understands the importance and necessity for following consistent, documented, and measurable processes in their work routines for continuous improvement.

A simplified view of the Credence quality process would be composed of the following major process steps:

- Definition of Situation
- Intermediate Fix
- Root Cause Analysis
- Corrective Action
- Evaluation and Follow-up

The goal of this process is to improve existing processes and more rapidly achieve conformance to requirements. A side benefit of course, is the fact that at the same time, the cost of reaching conformance is also minimized. Rather than trying to describe all the processes that are under scrutiny, this paper focuses on a single process that recently underwent significant revision.

II. The Process

Often processes attract attention to themselves through their failures, but other means of identifying processes in need of change also exist. Credence applies measures of Price Of Non-Conformance (PONC) and some Statistical Process Control (SPC) within the software engineering organization, although for the most part the use of these mechanisms is still fairly immature. As time goes on and more experience is built up within the organization, these mechanisms will play a larger role. In the meantime, one of the most effective means of identifying necessary process improvement areas has been the use of project post-mortems.

Around the completion time for any major project, a group meeting of all involved staff is arranged to discuss and brainstorm the key factors of success or failure in the process of doing that project. This isn't a new concept. It is well understood that people who are actually performing a process have a fairly clear view about which aspects of the process caused them the most work or which aspects seemed worthless or unclear. A project post-mortem taps into this knowledge base and extracts the key elements so that process improvements can actually take place. With changes coming from three or four major post-mortems in the past, it has become very clear that this is a crucial mechanism for indicating failures and successes.

The example that this paper addresses stemmed from one of the more recent post-mortem bullet items. It was found that software code reviewers had to view many different coding styles, often several of which appeared in any given module as responsibility for that module changed hands. This was considered to be severely detrimental to the readability of the software code and that work efficiency of the reviewers was lost. It was decided that an enforced software coding standard would reduce this problem to a negligible level.

A short term "fix" was established in order to gain experience with long-term possibilities. The Credence Standards Committee was chartered with the task of creating a corporate-wide software coding standard, and intermediate versions of the standard would be used as "working" standards by the committee members and some additional engineers. This provided an avenue for feedback about the workability of the standard. Although code generated during the time that the coding standard was in a state of flux may not necessarily meet the final standard precisely, it would still be of a significantly higher consistency and would require much less rework than code produced without any standard.

Additionally, some long-term process changes were also applied at this time in order to address other related process problems raised during the post-mortem. An example of one such process change was to have a "roll call" for reviewers to ensure that all reviewers present had sufficient time to prepare for the review. Unprepared reviewers cannot be permitted to attend a review since they will bog down the review process and provide only minimal value to the review. If an insufficient number of reviewers are prepared, the review is postponed.

Normally, the next step involves the identification of the actual source of problems, the root cause. In the example of a coding standard, the usual root cause analysis process step shrank dramatically because it was addressed in the initial situation assessment. The root cause of inconsistent code style was clearly a lack of coding standards and/or the lack of enforcement of said standards. Under other circumstances, analytical approaches may have been applied to establish those major root causes.

The corrective action phase encompasses the plan and execution of the process changes identified as those which would improve the existing process. The preliminary action is the establishment of a list of goals and limiting factors for the Standards Committee to work towards and within, respectively. In general, process changes in processes are identified, and new processes may be created or better detailed. For the software coding standard, this involved nailing down all the requirements that had to be addressed (changes in the check-in process, style issues, automatic documentation extraction, etc.) as well as the nuances that would assist in ensuring "buy-in" on the part of the engineering staff.

The Standards Committee was comprised of engineers from each of the three sites, but by far the largest number of members came from the Beaverton (6 members) site, with each of the other sites contributing one or occasionally two members. The committee met for an hour, once per week via conference call, to discuss the coding standard and other standards under development. This was considered to be "sufficiently intensive" to get the coding standard completed in a timely manner. Initial discussions about the standard began at the end of July 1992 with brainstorming sessions and a justification of the contents on the basis of comparison against the requirements. A fairly solid preliminary version of the standard was ready by early September.

The standard was distributed throughout the engineering community with a call for comments, and Standards Committee members started to adjust their work processes and styles to try out the utility of the standard. Very few responses came back, and the committee continued to tweak on the standard according to problems identified through the standards usage. In October, the first signs of problems made an appearance when a few engineers voiced their opinions to the committee, but an extended conversation seemed to clarify most of the issues. By late November, the coding standard was considered "complete" except for a few minor items, and some groups within Credence were already using it actively.

In late December the committee had nailed down the last few remaining issues and once again a copy of the coding standard was distributed to the engineering community at large with a call for final comments before it would become part of the corporate engineering process. With the minimal response that this action invoked, a trial period was announced and if no major problems the standard would take effect at the end of March 1993.

At this point, management stepped into the picture with the statement that conformance to this software coding standard was to become a work requirement and everything came apart at the seams. A group of engineers met over the course of several weeks and drafted a series of comments and concerns about the standard. The value of having a standard in the first place was questioned, as well as the use of an automated styling tool ("indent" switch settings are specified by the Credence coding standard) among other things.

Fortunately, beyond a few philosophical issues, most of the basic standard stood up under scrutiny and generally remained unchanged. The committee was able to meet and discuss the standard with those parties who were most concerned, and resolved many of the difficulties. A meeting of the entire software engineering staff was organized with a conference call to all sites. The standard went through one last review and the final issues were raised. These issues were discussed to try to reach a consensus, and if this was not possible the issue was put on a voting ballot. One week was given for people to voice their opinion and drum up support for their view before the voting was closed. The final memo to management with the final draft of the standard was written on July 1, 1993, almost a year after work on the standard had begun.

As the standard becomes a part of the common usage and people begin to use it, measures can be taken to show the extent of its reach. Simple measures such as compliance to the coding standard during code walkthroughs or reviews, and the percentage of the total code that complies with the standard, can be used to estimate how well the engineering community is adapting to using the coding standard. Over time, portions of the standard will probably display defects in application, and will require rework, but these should not have a major impact. Any major problems should have already been noticed during the extended trial use period, and only minor ones should remain. A project post-mortem of the process of designing the standard would also be a suitable activity.

III. The Post - Mortem

Although no official post-mortem process analysis has been completed at the time of this paper, most of the results of such an analysis are very clear.

A. The Key Risk Factors

The primary problem that the committee faced was inexperience in dealing with the wide variety of personalities that make up the Credence engineering organization. The scope of the problem never really struck home until those personalities were driven to action. This occurred at a fairly late stage in the course of events. In part, this failure came about because some people were not made aware that their individual work processes would be affected. In others, a sort of "ostrich maneuver" attitude kept people from acting. It was though they thought "If I don't say anything, maybe they will just give up and go away." Eventually, these people discovered that through their inaction, some things came to pass that they had lost the ability to avert. This is a clear example of a passive-aggressive behavior on the part of non-participants. This is very difficult to overcome or even recognize since with every standard sent out to review, little or no response was registered.

What use would a standard be without a mechanism to address emotional issues that always seem to appear whenever coding style is discussed? Examples of this sort of issue are the location of the curly-braces ("{ }") in a conditional statement, or the number of spaces for indentation levels. To resolve this so that the largest number of people would be satisfied, the use of the standard UNIX automatic styling tool "indent" was made a part of the standard. A select series of switch settings were chosen as the standard settings to ensure that code under source control wouldn't thrash from one format to another, but people could still use the "indent" facility to adjust the style of the code to match their own desires.

But this solution created other problems as well. Not all code is suitable for standard "indent" usage since it does not feature a "defeat" mechanism to protect carefully formatted columnarized data initializations. A GNU version of "indent" actually contains this feature. Additionally, some people had an aversion to adding a new process step, since they now had to actually run "indent" before check-in. This was addressed by encapsulation of new steps within existing steps by providing examples of aliases that run "indent" before check-in and after check-out.

Another risk is that of omission or inadvertently attempting to standardize something that shouldn't have been standardized. The likelihood that the standard will need immediate changes is rather slim, but at some time items may be identified as in need of change. Any standard of this type should be considered to be a "living" document, and should have a process by which it can be changed. As long as the Standards Committee exists, this avenue remains open. People need to be made aware that the committee has responsibility for the standard's contents, and that complaints and concerns are always accepted and given careful consideration.

Then there is the philosophical question of whether the coding standard limits creativity. This is largely an excuse used to avoid enforced order and the ever present aversion to change. Nothing within the standard addresses the functional breakdown or the organization of a body of code, those features which are most subject to creativity. It is solely a series of rules and regulations governing the layout and documentation of code to guarantee the readability and maintainability of the code.

B. The Key Success Factors

The number one success factor is the involvement of management as a enforcing agent for the software coding standard. Credence management stated "your conformance to a coding standard will be a mandatory work requirement." At the same time, management left it entirely to the committee to establish the contents of that coding standard. Once this was made clear to the entire engineering staff, participation in decision making and interest in the standard shot rapidly upward.

The Standards Committee was very secure in its belief that the coding standard was an important part of the set of improvements required by the Credence development process. The fact that the committee was empowered by management to make all the guiding decisions and establish the rules for the standard allowed the committee to make decisions requiring no higher authority. The committee acted as facilitators for process improvement.

Management allowed one other crucial aspect. There was no hard deadline for completion of the standard. The coding standard is a set of rules which requires behavioral changes. These changes can only be achieved by a corporate-wide buy-in of the standard, a sort of global "mind-alignment" of the engineering staff. Management gave the Standards Committee sufficient time to go through a process of conflict handling.

IV. Analysis

Things do not always go according to plan. In fact, when it comes to dealing with any moderately sized group of people (25-40) it is difficult to compensate for both "group effects" and "individual effects." In the case of Credence, the three geographically separated groups of engineers at each of the major sites were further separated by their history of organization and operation. Within each site, individual desires, as well as established modus operandi have to be carefully dealt with, for they are a portion of those factors which contribute to the freely chosen adoption of the standard.

From studies on organizational behavior[1] there are several modes people fall into in dealing with situations involving conflict.

- avoidance
- competition
- accommodation
- compromise
- collaboration

Aspects of each of these appeared over the time of the standard's development. The committee was able to work with them to varying degrees of success.

Avoidance was practiced both by the committee as well as those people who assumed that the standard would never really materialize. The committee should probably take the blame for not recognizing the significance of non-participation by outside groups. The composition of the committee should also have been more balanced across the three sites instead of the 6-1-1 ratio. But not all the blame should be leveled at the committee, because the committee has always been open for additional members, but participants primarily volunteered from one site.

Competition did not play as large a role, although it was personified by oratorical lectures pushing one or another view point. The competition was about choosing among multiple possibilities for the standards. As time progressed and it became more obvious that there was intentional lack of participation or stalling actions, patience occasionally wore thin, and there was an urge to "slam the door" on late-arrivals. For the most part however, this urge was suppressed before it caused a problem.

Accommodation is very important in defining a coding standard. It is the opposite of restriction. Rather than potentially restraining creativity, a coding standard needs to be written to allow as many options as possible. Similarly, the committee's actions were directed to accommodate anyone who had a justifiable request.

Compromise is necessary in any activity that involves "mind-alignment," and the standard is no different. Almost every line item in the standard is a compromise of one sort or another, if they weren't they wouldn't need to be listed in the standard, they would be followed anyhow. As much as possible, verbal compromises were made, and voting was avoided. In the end, only three fairly inconsequential standards had to be voted on. Voting should be considered a last resort in conflict resolution because it inherently involves one side winning and the other losing.

Collaboration is achieved by focusing attention on meeting the specified goals and staying away from personal interests. Maintaining a desired level of collaboration is dependent on the groups ability to retain cohesion; belonging to the group should be viewed as desirable by its members. However, closely knit groups run a higher risk of "groupthink", so some mutual aggression within the group is also desirable. Note how this ties back to competition through aggression. In turn, this aggression is fed by the understanding that the committee has been given the authority to set the rules by which its members will work under during the other aspects of their job.

V. Conclusion

Certain factors made this software coding standard possible, but the most important of these factors was that of management commitment to a process of quality improvement. This factor is strictly the responsibility of management, especially upper management.

Many companies embark on a "Commitment to Quality" program by delegating the task. They simply assign someone to "improve quality". This approach is not good enough because the people assigned the task of quality improvement lack the authority to commit sufficient resources to the task. True management commitment contains the following aspects:

- Active participation and interest by top management in all aspects of quality improvement.
- Allocation of financial, time and personnel resources to quality improvement.
- Assignment of accountability to individuals involved in quality improvement as a work requirement.

Credence management has adopted an approach which focuses on participation in quality improvement. This closely approximates a Theory Y model. McGregor's Theory X vs. Theory Y outlines the difference between an organization which operates under authoritarian management (Theory X) and one which operates under participative management (Theory Y).[2] Theory Y identifies with the fact that people desire involvement and wish to grow with an organization, both

inwardly, with confidence and security, and outwardly with new skills and responsibility. Credence management has enabled the Standards Committee to operate with sufficient autonomy to create a standard that can be used throughout the corporation. At the same time the management also provided the authoritative backbone for the standard to be taken seriously. Without each of these two aspects, the standard would be a failure.

In retrospect, one thing the Standards Committee needs to do differently in the future is to better utilize management support. The support does not have to be justified, as it is a component of Credence's quality commitment. Requesting a verbal statement from management along the lines of "You will follow a coding standard," should have been done at an earlier point in time. This acted as a trigger to attract participation on the part of the entire software engineering staff, and thus build the long-term buy-in.

VI. Bibliography

1. Bowditch, James L./ Buono, Anthony F., "A Primer on Organizational Behavior", 1990, John Wiley & Sons, NY. pp. 147-148.

2. McGregor, Douglas, "The Human Side of Enterprise", 1960, McGraw-Hill, NY.

FOCS: A Classification System for Software Reuse

Jürgen Börstler
Lehrstuhl für Informatik III,
Aachen University of Technology,
Ahornstraße 55, D-52074 Aachen, Germany
e-mail: jubo@rwthi3.informatik.rwth-aachen.de
phone: +49 / 241 / 80-21320
fax: +49 / 241 / 80-21329

Abstract

Reuse is one of the key technologies leading to the production of cheaper and more reliable software. In this paper we introduce feature-oriented classification, and its usage to describe and structure the properties of documents, as well as to support their storage and retrieval.

In feature-oriented classification documents are described by sets of features. Each feature describes a single property of the document. The semantics of a feature is given through its successive refinements, until well-known notions are reached. The refinements span trees. Each of these trees contains features corresponding to the same aspect, similar to a facet, as used in faceted classification. We distinguish two different kinds of refinement, in order to support the understanding and structuring of the features. The user can query the library with a set of features. The System will then return all documents which possess these features. To support the retrieval of similar documents we define a similarity metric between features.

Since classification schemes tend to change as knowledge on the covered domains increases, we allow for arbitrary changes to the classification scheme. Some of these changes invalidate existing descriptions of documents. We will discuss how FOCS supports consistency management in such cases.

Keywords/Phrases: Design for Reuse, Design with Reuse, Impact of Reuse on Quality, Libraries, Tools

Biography

Jürgen Börstler was born in Germany in 1960. He received the M.S. degree from Saarland University, Saarbrücken, Germany, in 1987. He will receive the Ph.D. degree from Aachen University of Technology in 1993. His dissertation is on design languages and design tools with special respect to reuse.

Since 1987 he is a Research Assistant and Lecturer at Aachen University of Technology, Germany. His current research interests are reusability, (object-oriented) design, software development environments, and traceability between lifecycle phases.

Mr. Börstler is a member of the IEEE Computer Society, the Technical Committee for Software Engineering, and the Association for Computing Machinery.

FOCS: A Classification System for Software Reuse

Jürgen Börstler
Aachen University of Technology, Germany

1 Introduction

Reuse is one of the key technologies leading to the production of cheaper and more reliable software. The literature also reports a strong correlation between high reuse rates and low fault rates of software components (/CCA 86/). Approaches to reuse can be seen from two different points of view:

Development for reuse addresses the fact that documents[1] are not reusable per se. Documents must have been developed for later reuse. Therefore the documents may need to be generalized to be applicable in a wide range of applications. A document itself supports reuse, if it encapsulates a single, well-defined abstraction, hides the details of its realization, and fulfills some predefined quality criteria. It should clearly list its dependencies on other documents, and/or the environment, but there should be as few as possible such dependencies. To say it in other words, the application of the well known (design) principles, like abstraction, information hiding, encapsulation, structuring, parameterization, coupling, and cohesion is critical for reuse. To support development for reuse modern design languages are necessary to describe reusable components. To handle and manipulate documents written in such a language one needs sophisticated tools, like syntax directed, context sensitive editors, analysers, transformers, etc. (/Börs 93/). Integrating such tools into a modern software development environment is a great step forward to make reuse work (/Börs 91a/).

This paper mainly addresses **development with reuse**, i.e. the (re-)usage of predefined documents to build new applications. The usage of libraries to store all kinds of reusable documents is one promising approach to aid development with reuse. But most of these libraries did not succeed as expected. The designers of such libraries noticed that it is not enough to add great bunches of code to the libraries, and leave the (re-)users alone with it. Therefore, additional tools were developed to describe, store, and recover the stored documents. Classification is an important tool supporting this kind of approach (e.g. /PrFr 87/). We therefore propose FOCS (**f**eature-**o**riented **c**lassification **s**cheme), a new approach to the classification and retrieval of documents.

For the organization of complex domains controlled vocabularies (idexes) have a long and successful tradition (/Linné 58/) for classification. Since documents usually have orthogonal aspects describing their properties the classification of documents according to more than one aspect (facet) in parallel, seems appropriate. There exist several recent approaches in this direction (/PrFr 87/, /Fra 89/, /HeFr 92/, /KST 92/). Most of these approaches rely on static classification schemes and/or only consider functional properties of the stored documents. Our approach allows for changes in the clas-

1. Documents can be all kinds of (partial) results produced during the life-cycle of (software) systems, like requirements definitions, designs, code, test cases, and so on.

sification scheme. We think this is important, since classification schemes tend to change as knowledge on the domains covered increases. Our classification scheme can also deal with the classification of structural properties, as necessary for data-oriented documents.

The rest of the paper is organized as follows: In chapter 2 we introduce FOCS using several examples. In chapter 3 we describe how this approach can be used to describe, store, and recover documents. Chapter 4 deals with changing classification schemes, and how the document library can be kept consistent with these changes. In chapter 5 we describe how integrated software projects support environments effect reuse and quality throughout the lifecycle. A conclusion follows in chapter 6.

2 Feature-Oriented Classification

In feature-oriented classification documents are classified by sets of features. Each feature describes a property of the document. We call such sets of features descriptors of documents. To support the understanding and construction of descriptors the features are organized in a classification scheme. The semantics of features are given through successive refinements, until well-known notions are reached. These well-known notions correspond to the attributes or terms known from other classification schemes. The refinements span trees. Each of these trees contains features corresponding to the same aspect, similar to a facet, as used in faceted classification (/PrFr 87/). A feature can be interpreted as a path in a facet tree. Figures 1 to 4 show an example classification

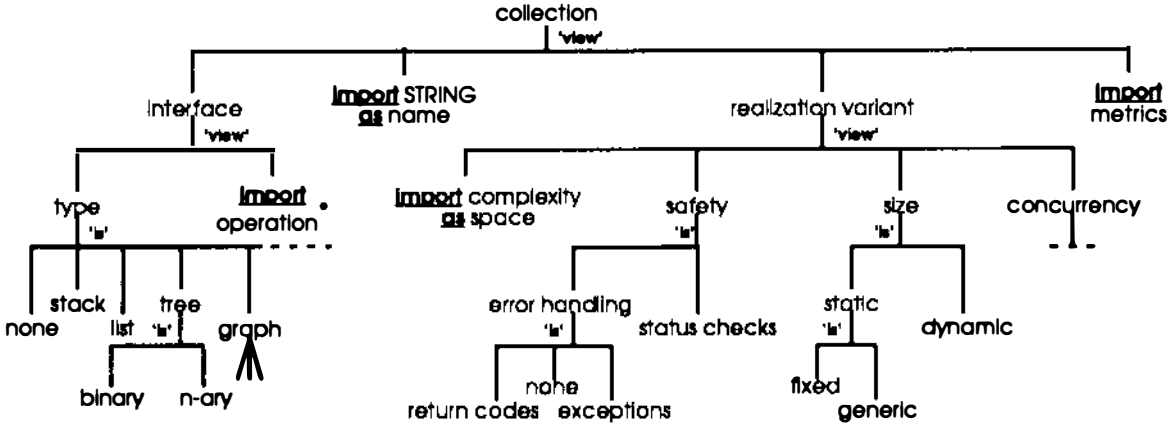


Figure 1: Part of an example classification scheme (facet tree collection).

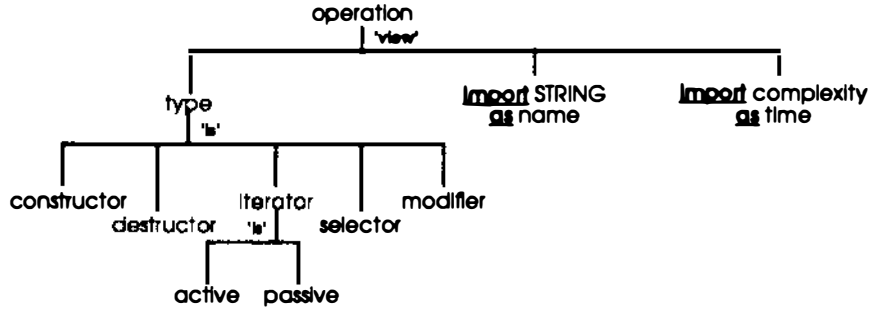


Figure 2: Facet tree operation of our example classification scheme.

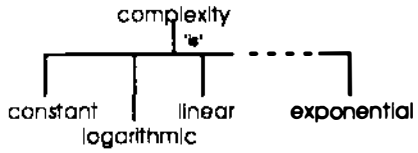


Figure 3: Facet tree complexity of our example classification scheme.

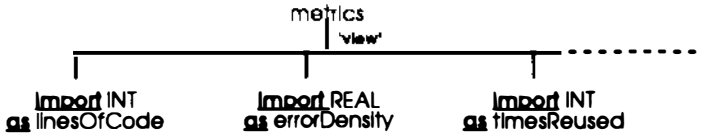


Figure 4: Facet tree metrics of our example classification scheme.

scheme for the usual collection types/objects. According to this classification scheme a stack type component may be described by the following descriptor:

(collection→name→Example,
collection→interface→type→stack,
collection→realization variant→space→constant,
collection→realization variant→safety→status checks,
collection→realization variant→size→static→fixed)

We support two different relationships, representing different semantics of refinement:

(a) A **view**-refinement represents a distinguishable aspect of a feature. Each document may be classified alongeach view of a feature. In our example a collection can be classified according to its interface, and realization variants.

(b) An **is**-refinement represents the specialization of a feature. Each document may be classified by only one specialization of a feature. In our example the type of an interface can either be a stack, a list, a tree, and so on.

To avoid the duplication of work we support the **importation** of facet trees into other facet trees. To assign the imported subtree an appropriate name, importation is combined with renaming. The star (*) denotes repetition of a feature. Repetition is necessary to allow for multiple occurences of a feature in the same descriptor. According to our example classification schemes descriptors may contain more than one occurrence of the feature operation or its refinements. Using repetition each operation of our example component can be described. To relate the refinements belonging to the same occurrence of a feature these refinements must be grouped to subdescriptors. The groupings form nested subdescriptors (see descriptor below).

We support build-in schemes for INT, REAL, and STRING. These build-in schemes can be treated as implicitly is-refined into all possible elements.

We do not require a complete classification of documents. We allow descriptors to contain non-leaf features, as well as to completely omit the classification along some facet trees. This is useful in sevaral ways:

- Some facet trees or views may only make sense for special kinds of documents.
- There may exist components, which can be described with significant detail, without using all facet trees or views.
- We want to support a step by step evolution of descriptors.

We therefore demand for correct descriptors only. There is no reason for the rejection of an uncomplete descriptor, since it may be sufficient to gain an useful retrieval result. Nevertheless, it should be clear that a more detailed descriptor should be preferred, since it contains more detailed information about a document.

A detailed descriptor of our example component may the look like:

```
( collection→name→Example,
  collection→interface→type→stack,
  collection→interface→operation→
    (type→constructor, name→push, time→constant),
  collection→interface→operation→
    (type→destructor, name→pop, time→constant),
  collection→interface→operation→
    (type→selector, name→top, time→constant),
  collection→realization variant→space→constant,
  collection→realization variant→safety→status checks,
  collection→realization variant→size→static→fixed,
  collection→metrics→linesOfCode→250,
  collection→metrics→errorDensity→0.01,
  collection→metrics→timesReused→10 )
```

We also support cross-reference links to guide users through the classification schemes.

3 Using the Classification Scheme

The nodes of the facet trees contain pointers to documents satisfying the feature corresponding to this node. Note that all refinements of a feature also satisfy the refined feature. Therefore, all nodes of a facet tree have pointers to all documents satisfying at least one of the features corresponding to its subtree, as shown in figure 5. To identify

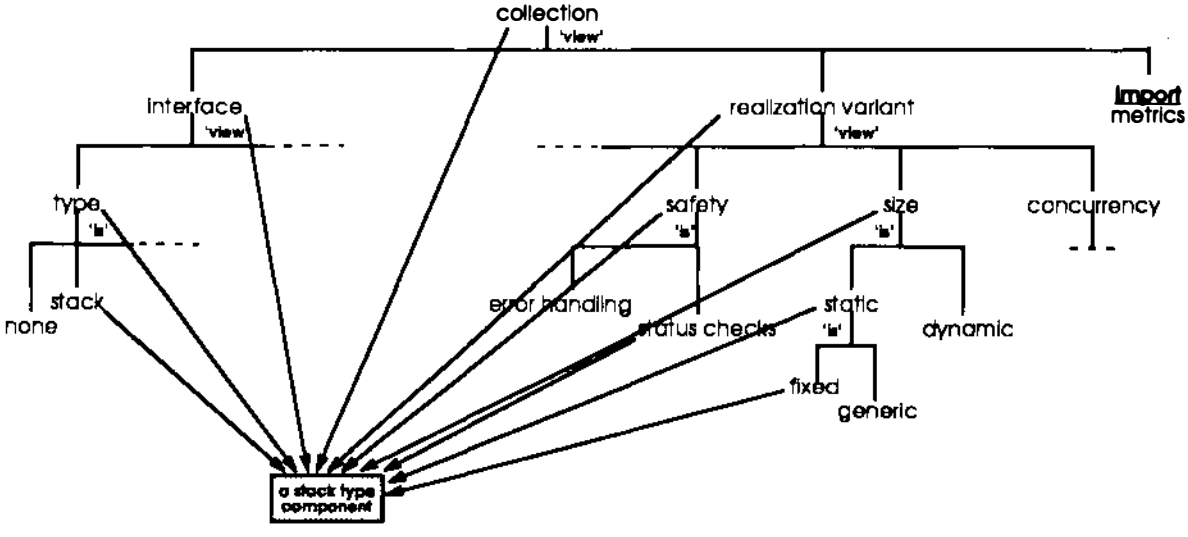


Figure 5: Example classification scheme with pointers to example stack type component (excerpt).

the document pointers constituting a subdescriptor, we introduced special helper nodes not shown in figure 5. This is necessary for an unambiguous identification of documents. For more details see /Börs 93/.

Retrieval can then be supported by browsing, or by providing a descriptor. The browsing capabilities may be enhanced by the introduction of additional cross-reference links connecting related features.

If one has only a **vague idea** of the features of a document, one can start **navigating** through the classification scheme beginning at an appropriate (sub)tree (facet). Since is-refinement represents feature specialization, navigation along a is-refinement of a feature will reduce the number of corresponding documents (matches). Navigation to a view-refinement will focus on a special aspect of the same set of documents. Since all matching documents can be reached from any node in the classification scheme, detailed information on each document can be displayed. Users may introduce their own cross-references to circumvent long navigation paths.

If one **knows some features** a document requires, one can construct an appropriate descriptor. The system will then return all documents **matching** this descriptor. The construction of a descriptor can be done manually, or by selecting a set of features in the classification scheme.

In general one cannot expect to find a document, which exactly fulfils the needs, i.e. exactly matches the given descriptor. In this case the user wants to be provided with a set of documents close to his needs. To compute such a **similarity metric** the matching algorithm takes into account the structure of the classification scheme. The closer two features are placed in the classification scheme, the more similar they are and the more similar are the documents providing these features. According to this binary trees and n-ary trees are more similar, than e.g. binary trees and graphs. Unfortunately, our classification scheme only provides very weak information on the similarity of adjacent nodes related by a refinement step, e.g. graph, tree, list, ... Since only is-refinements do restrict the search space, it is sufficient to consider them only.

To improve the usefulness of the similarity metric we introduced the **satisfies**-relationship (see figure 6). A feature f_1 satisfies a feature f_2, if a user searching for a

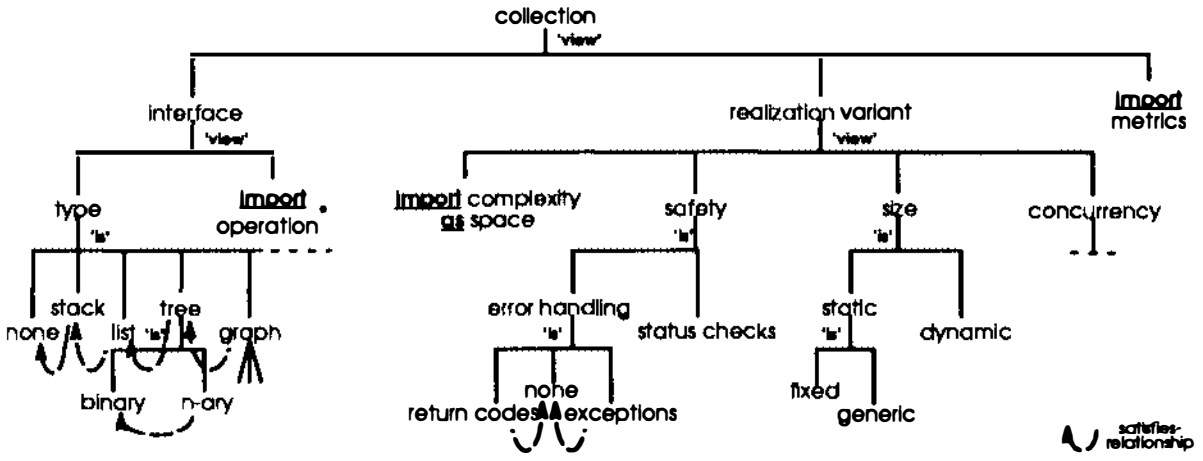


Figure 6: Introducing the satisfies-relationship into our example classification scheme (excerpt).

document with property f_2, will be satisfied by receiving a document fulfilling f_1. The satisfies-relationship is reflexive and transitive. Nevertheless, the closer two features are related with respect to this relationship the more similar they are treated, i.e. list and tree are more similar than list and graph. Therefore there must not be a cycle in this relationship. This property is checked by our editor for classification schemes.

For each feature f in the descriptor all documents classified by features satisfied by f, or one of f's is-refinements will be returned. Is-refinements inherit the satisfies-relationships from their predecessors, i.e. ...→type→tree→binary satisfies ...→type→list, since ...→type→tree satisfies ...→type→list. Note that a simple ordering, e.g. from left to right is not sufficient, since the satisfies-relationship is a partial order only (see safety subtree for example). For our build-in schemes INT and REAL the usual '≤' is defined as the default satisfies-relationship. For the build-in scheme STRING we have not defined any satisfies-relationship, since a default strategy like lexical order does not make sense for the computation of a similarity between documents.

In our example scheme from figure 1 a descriptor containing the feature tree will yield all documents referred to from node tree, as well as those referred to from binary, and n-ary. This set of documents will then be merged with the result built so far. Merging can be done in several modes. The 'exact-match' mode will simply perform a set intersection. Other modes may relax descriptors, e.g. by replacing a feature by the features it satisfies, or its is-generalization. In addition, features can be given weights to indicate their relative importance in a descriptor. These weights can also be considered by the matching algorithm.

4 Maintaining the Classification Scheme

Since classification schemes tend to change, we built a syntax-directed, context sensitive editor for classification schemes. This editor also provides support for analyzing documents in the underlying language, i.e. classification schemes. In some cases, changes to the classification scheme may invalidate existing descriptors. Since our classification scheme contains references to the documents as described before, we have immediate access to all documents corresponding to descriptors possibly affected by a change to the classification scheme. We will now discuss how we can react to such changes. To simplify the discussion we distinguish between several kinds of such changes:

(a) **Non-structural changes** to the classification scheme (e.g. renaming a feature, or introducing non-hierarchical relationships).

Since our editor guarantees, that new names do not conflict with already existing ones, an update can be performed automatically. Since non-hierarchical relationships do not effect descriptor construction these modifications can be made freely.

(b) **Simple structural changes** to the classification scheme (insertion/deletion of leaf nodes, e.g. refining a feature).

Insertion of a new refinement offers the opportunity of more detailed classifications. Since our editor guarantees that extensions do not conflict with the classification scheme built so far, all descriptors remain correct. Although such modifications do not cause problems, it is important to propagate the updates regularly.

Deletion of a leaf node invalidates all descriptors containing the feature corresponding to this node. To correct this we can automatically replace this feature by its generalization. These corrections may produce descriptors containing useless features, i.e. features, which do not narrow the search space.

(c) **Complex structural changes** to the classification scheme (structural changes to non-leaf nodes, e.g. deleting a refined feature, or moving subtrees).

This category contains the most critical changes. Deleting a refined feature may not only invalidate all descriptors containing this feature, but also those which contain refinements of this feature. The simplest way to correct this is to remove the whole subtree rooting at this feature and proceed as above. But in most cases we want to preserve the document pointers in this subtree (see figure 7). In most cases

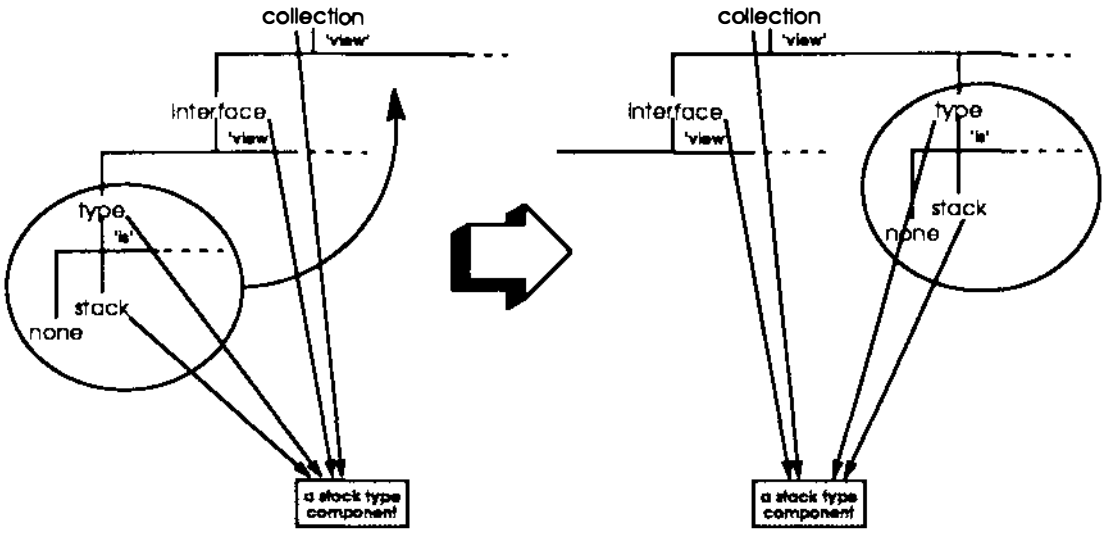


Figure 7: Example for a complex structural change in the classification scheme (moving the subtree rooted at type).

these changes can be done automatically. Problems may occur in the case of (nested) subdescriptors. These are discussed in detail in /Börs 93/.

Although we allow for arbitrary changes in the classification scheme, complex changes should not happen very often. Complex changes do not only affect descriptors of classified documents, but also the descriptors used to search for documents. The same descriptor may yield a completely different result after a change in the classification scheme has occurred. This will confuse users of the classification scheme. Changes should therefore not only be syntactically safe as described above, but also semantically safe, i. e. only slightly effect the result yielded by a user query.

5 Integration of FOCS into IPSEN

The IPSEN prototype developed at our department is a comprehensive set of cooperating tools, which support the development and maintenance of software (/ELNSS 92/). The documents produced during software development represent the software project at different levels of abstraction, and in different degrees of formality. The contents of these documents are therefore highly interrelated. These relationships can be established between increments of one document, as well as different documents. We even support the definition of relationships between increments of different document types. Such relationships are used for example to connect increments in a requirements definition to their corresponding counterparts in design documents to support traceability (/BöJa 92/). These relationships are realized as hypertext-like links, which are typed and may have different semantics dependent on the (type of) linked incre-

ments. We also support course-grained links to relate complex objects (which may themselves include other links). In this way a software project can be described as an integrated set of (consistent) documents.

IPSEN is designed as an **integration framework** (/Lef 93/). We have developed a set of guidelines to create new tools and integrate them into IPSEN without (major) modifications to already existing ones.

All IPSEN tools share the same characteristics. They operate on a document through a unified user interface. We maintain a set of current increments (selectable parts of documents) and commands are filtered. In this way the user can only execute commands he is allowed to apply and which are applicable on the selected increments. After execution of a command all affected documents are updated incrementally. For the sake of efficiency these updates may be buffered and performed on demand by the users. The updates also consider existing links in and between different documents. A snapshot of a set of interrelated documents from the tools-users point of view is shown in figure 8.

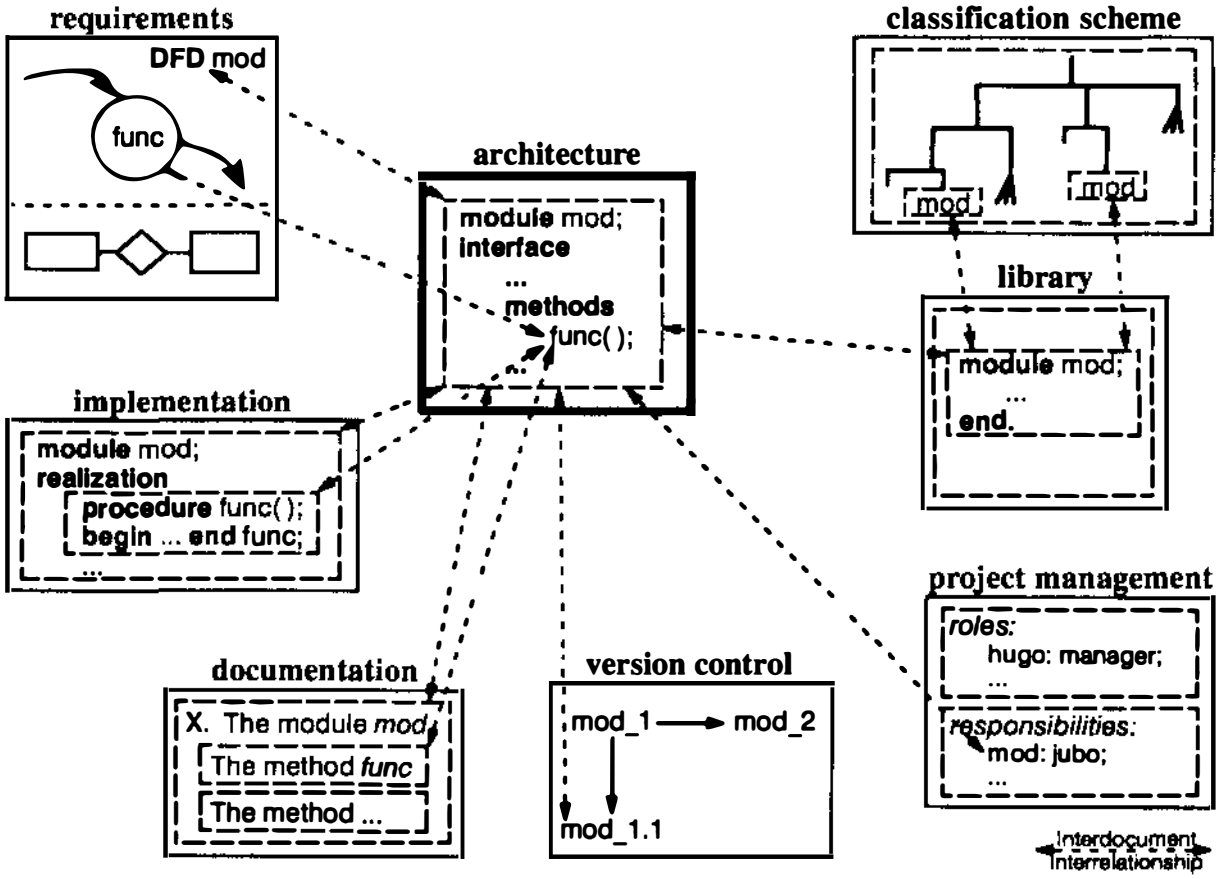


Figure 8: Integration of documents in IPSEN.

This model of software development prevents errors as early as possible. Errors detected in later stages of the development process, can be traced back to their origin. Therefore, the errors themselves can be corrected, instead of their symptoms. Quality is built into the development process, instead of applying back-end processes.

To benefit from libraries it is not only important to find an appropriate document, but also to understand it, and to adapt it to the problem at hand. In IPSEN architec-

tural design plays an important role to overcome these problems. This is due to the fact that most documents depend (directly or indirectly) on the software architecture document, which is created during architectural design. The design editor in IPSEN gives support in this direction. It supports a language which offers almost all facilities important for development for reuse. In addition we have developed and implemented a set of guidelines for the construction of interfaces of data abstraction modules (/Börs 91b/).

A tight integration with the design editor makes it possible to implement an active reuse tool. By now reuse tools require the designers to explicitly query a library, know a lot of rules and guidelines how to build architectural designs, and integrate recovered documents by hand. If the reuse tool is integrated with the design editor, it may constantly search the library for documents matching parts of the current design. After a match the reuse tool may then go into a questionnaire to find out if the recovered document fulfills the designers needs. It may then also help in adaption and integration of documents by applying appropriate transformations to them.

6 Conclusion and Future Work

In this paper we described FOCS, a new classification scheme to support development with reuse. We have described how sets of features can be used to support the description, storage, and retrieval of documents. The features are organized in classification schemes to indicate their meaning, and their interrelationships. Using these classification schemes we can compute a similarity metric between documents, to support the retrieval of similar documents. We also discussed how this scheme can be used to implement a browser for the library.

Compared to other classification schemes (/PrFr 87/, /Fra 89/, /HeFr 92/, /KST 92/) FOCS is applicable to a wider range of of applications, since features are well suited to describe all kinds of properties of various kinds of documents. FOCS is also very flexible in allowing smooth changes to the classification scheme without corrupting the library.

We have already built a small prototype for FOCS. It includes a syntax-directed editor to create and maintain classification schemes, a simple library to store documents and descriptors, as well as a matching algorithm, which already takes into account weighted features. The prototype supports some of the changes discussed in chapter 4. It does not support complex structural changes to the classification scheme. In this cases most of the document pointers have to be readjusted manually. Better maintenance support is planned for further prototypes. In the future we want to integrate this prototype into the IPSEN system. A lot of work is still necessary to provide designers with an active reuse assistant as described above.

Although we only focused on classification, there are other working areas where reuse must be considered. In IPSEN there are two such areas, where reuse will be examined in future. The first one is requirements specification. Besides the specification of functional and data-oriented aspects of requirements, we have integrated real-time specifications into our requirements specification language (/Beeck 92/). These real-time specifications can occur at different levels of abstraction. The lower levels serve as abstract behavioral specifications of families of requirements, which can be reused to define detailed requirements (/GaFr 90/). The other point of interest is the incorporation and reuse of knowledge into the transformation process from requirements to design as described in /Lef 93/ and /BöJa 92/.

References

/Beeck 92/ M. von der Beeck, "Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification," Technical Report AIB Nr. 92–26, AachenUniversity of Technology, Aachen, Germany, 1992.

/BöJa 92/ J. Börstler and T. Janning, "Traceability Between Requirements and Design: A Transformational Approach," *Proceedings COMPSAC '92*, Chicago, Ill, Sep 21-25, 1992, 362–368.

/Börs 91a/ J. Börstler, "Integrating Reuse into a Software Development Environment," *Proceedings of the First International Workshop on Software Reusability*, Dortmund, Germany, July 3–5, 1991, 234–238.

/Börs 91b/ J. Börstler, "An Editor for Programming-in-the-Large as a Central CASE-Tool," in German, *Proceedings TOOL '91*, Karlsruhe, FRG, Nov 26–28, 1991, 341–350.

/Börs 93/ J. Börstler, *Programming-in-the-Large: A Language, Tools, Reusability*, PhD Dissertation (in German, forthcoming), Aachen University of Technology, Aachen, Germany, 1993.

/CCA 86/ D.N. Card, V.E. Church, W.W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, Vol. SE–12 (2), Feb 1986, 264–271.

/ELNSS 92/ G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, A. Schürr, "Building Integrated Software Development Environments Part I: Tool Specification," *ACM Transactions on Software Engineering and Methodology*, Vol. 1 (2), Apr 1992, 135–167.

/Fra 89/ W.B. Frakes, "Panel Session: Information Retrieval and Software Reuse," *Proceedings SIGIR '89*, Cambridge, MA, Jun (25–28) 1989, 251–256.

/GaFr 90/ A. Gabrielian and M.K. Franklin, "Multi-Level Specification and Verification of Real-Time Software," *Proceedings of the 12th International Conference on Software Engineering*, Sep 1990.

/HeFr 92/ B. Henderson-Sellers, P. Freeman, "Cataloguing and Classification for Object Libraries," *ACM SIGSoft*, Vol.7 (1), Jan 1992, 62–64.

/KST 92/ E.-A. Karlsson, S. Sørumgård, Eirik Tryggeseth, "Classification of Object-Oriented Software for Reuse," *Proceedings TOOLS Europe '92*, Dortmund, Germany, Prentice Hall, 1992.

/Lef 93/ M. Lefering, "An Incremental Integration Tool Between Requirements Engineering and Programming-in-the-Large," *Proceedings RE '93*, San Diego, CA, Jan 1993, 82–89.

/Linné 58/ K. von Linné, "Caroli Linnaei systema naturae, per regna tria naturae, secundum classes, ordines, genera, species," Thoma Nob. de Trattnern, Vindobonae, 1758.

/PrFr 87/ R. Prieto-Diáz, P. Freeman, "Classifying Software for Reusability," *IEEE Software*, Vol. 4 (1), Jan 1987, 6–16.

Conclusions from an industrial consulting project towards a return of effort model for software reuse

THOMAS GRECHENIG STEFAN BIFFL

Department of Software Engineering,
Vienna University of Technology,
Resselgasse 3/2/188, Vienna,
A-1040 Austria

EMail: Grechenig@eimoni.tuwien.ac.at
Tel.: ++43-1-58801-4082
Fax: ++43-1-504 15 80

Abstract. The following paper deals with experiences derived from an industrial consulting project for designing a reuse model. The model is oriented towards actual use in practice and therefore also towards different levels of reuse intensity. It was outlined for a company employing more than 200 software engineers who primarily develop software for bank services and bank administrations. The field of investigation on hand is connected with a lot of vital software engineering (SE) questions for any larger developer: heterogeneity in age of software, heterogeneity of applications, heterogeneity of development environments as well as different levels of software engineering consciousness and of knowledge among the software engineers. The model presented is drawn from state-of-the-art suggestions in reuse research which were adapted to meet local constraints of time and costs. The model can be taken as a recipe for reuse in practice as it is providing three different levels of reuse intensities/investments, and thus returning three different levels of reuse maturity.
Level I reuse maturity in practice is to achieve maintainability: Many older programs turned out to be widely undocumented; often requirements and/or abstract design were missing, the programs do not meet basic criteria of maintainability.
Level II reuse maturity is represented by balance within similar projects: We define a group of software systems as balanced, if there is a clear top-down structure from the general to the specific in documents concerning analysis, design, code and test. A new but similar system can be designed reusing upper level software document components and adapting lower level ones.
Level III reuse maturity affords several technical and organizational efforts to establish a true reuse culture. Making a reuse culture work needs developing, providing and enforcing of standards. On the technical level this requires the use of repositories for all phases of development as well as the application of quality assurance methodology. On the organizational level cooperation among people responsible for the development of standards (metric analysis, quality check through reviews) as well as those coordinating the reuse environment is required. The roles for a reuse culture as well as the educational prerequisites are defined.
The value of the presented work lies in mapping the reuse state-of-the-art to often appearing financial and organizational restrictions consciously, thus opening possibilities beyond the "total reuse or no reuse" advice.

Keywords: software reuse, reuse model in practice, industrial experience, maintainability, standards, quality assurance, management issues

The authors

Stefan Biffl and Thomas Grechenig are assistant professors at the department of software engineering at the Technical University of Vienna, Austria. They both received MS degrees in computer science and Ph.D. degrees in software engineering. Stefan Biffl teaches quality assurance and has a special interest in code metrics. Thomas Grechenig is doing research in requirements engineering and user interface design. They are both involved in organizing a software engineering workshop for 500 students. They have experiences as consultants in industrial (research) projects concerning quality assurance and reuse of software.

1 Introduction

Finding an appropriate and tested procedure within a pool of available software, instead of "inventing" it anew, is the idea of any reuse activity. The reuse of software documents has been suggested since the early eighties. Many suggestions have been made, some industrial experiences have been reported, tools have been designed so far (see chapter 2), such as repositories, CASE-tools, or the object-oriented paradigm, which all promise and proclaim some kind of reuse. Beyond these efforts it is obvious that successful reuse is a goal that requires technical and organizational preconditions: an ideal environment for reuse is more or less equal to an environment necessary for an ideal process of software engineering. Excellent software engineering will necessarily result in reusable software (because a fully conscious approach will anticipate many future uses and changes). Nevertheless it is a common experience that the everyday industrial development is different. There is no time and money for redesign, the requirements are incomplete, documents are already inconsistent within one project, design gets lost over time.
This article is about experiences in implementing reuse activities within an industrial environment. It can be regarded as a case study answering the question: What can reuse theory provide within reasonable time/cost constraints once you come across an existing software development situation? "Reasonable" in the context on hand means primarily the cost an average middle manager can be persuaded to invest. Within the decision space of this person (see fig. 5) any large investment proposed by a person of engineering competence is regarded as mere technical purity and aesthetics.
The reported research project is a supervision of a general company's effort towards a fundamental change of software structure, making a shift from a mainly mainframe-oriented software development and data processing structure to a dominant off-line structure. Within this new concept applications will run on workstations and central processing will be limited to data transactions.
Apart from the change in hardware configuration and hardware clustering towards a client-server-host architecture the whole software development strategy has to be redesigned. The object-oriented paradigm has been chosen to be the future development approach for front-end and server applications. Academic consulting has been required for the introduction of OO and reuse. The client was sure that OO development does not achieve reuse by itself. Because of the business area on hand the following constraints are important: independence from other software suppliers, continuity of development environment support, secrecy of actual software system details.
The goal of the paper is to report on conclusions for introducing software reuse in practice drawn from our consulting experiences. The second chapter provides a survey on the various reuse tools, concepts, and methods. Chapter three outlines intensity level I and II of reuse in practice. Level I is represented by reuse within one single project. Level II reuse can be established within a group of similar projects (a branch). In chapter 4 level III reuse is proposed as a (rather) ideal reuse culture. Level I is equivalent to maintainability. Level III includes definition and assurance of standards as well as a maximum of document reuse in any phase of development. The second level includes an effort towards a domain-restricted balance of projects lacking a special reuse and standardization management.

2 A Survey on Approaches to Reuse

2.1 General Remarks and Experiences

Productivity gains and quality improvement are at the core of effects expected from reusing software products. Moreover, reuse seems to be simple in the eyes of the beginner:

> "*Just take a piece of existing software and put it into a new product; that's what all good programmers do anyway*" (see [23]).

There are well known difficulties with this sort of naive reuse: There is usually a lack of "critical mass" of reusable components to start spontaneous reuse and keep it up as a regular practice. Normally the management is not willing to take the charge of necessary initial investments to help create reusable parts, and last but not least the average programmer seldom trusts of other people's code (see [8]).
The same author states that reuse success needs management commitment to the proper capitalization of software engineering (SE) infrastructure as well as organizational standards for design, programming, and reuse activities. [15] observes that all successful reuse projects had support from upper management. The working of reuse concepts needs a sound SE framework as basis; even extensive reuse investments will not cure a too-immature SE environment and will fail unless a certain level of consciousness is attained.

2.2 Some Reports of Industrial Reuse Efforts

[13] early reported on an insurance reuse project. The goal was to promote the reuse of COBOL code by supporting the research on the exchange of code, training, and support to eliminate highly redundant steps during code creation. The assessment and communication of code is made by a Reusable Code Review Board which consists of one member per application programming division.
[19] reports on a reuse level of 60% to 80% of COBOL source code lines by reducing the usual redundancy of code in commercial programs through employment of generic code-skeletons.
[11] describes The Reusable Software Library (RSL) a component library where code fragments labeled with keywords can be retrieved. He states that up to a third of the code can be reused trading off higher productivity in programming for lower performance of the resulting program.
[30] analyzed a set of moderate to large projects for unmanned spacecraft control (Quantitative Study of Spacecraft Control Reuse at GSFC). He found out that on the average a third of the source code could be reused or modified from previous projects. The modules that were reused without revision turned out to have less interaction with other modules or the user, to have simpler interfaces and more comments per source line than modules which were newly developed or revised.
The effort of the SEI domain analysis experiment had as a result that proper domain scoping is critical to success, there are no global but only local domain experts, and model representations may be domain-specific as well.
All case studies deal with quite specific environments and focus primarily on the reuse of source-code. The reported degree of possible code reuse depends on the diversity of the domain and ranges from 30% (broad domain) to 70% (narrow domain). Wherever authors claim to do a certain document reuse, the detailed working conditions for that reuse are missing. The guidelines which are applicable in a more general way resemble general wisdom of good SE practice.

2.3 Current Approaches in Practice

A company-wide reuse program is set up to improve the productivity and the quality of software production starting off from an already well-working SE environment. Thus a program of that kind will need more extra training and stimulus of the participants than new fancy tools. [33] underlines the importance of incentives to raise personal efforts for reuse activities: He suggests the assignment of the creator's name to a reuse unit as well as cash rewards for good reuse ideas or the recognition of reuse issues by management and the inclusion of reuse actions in the design and the project debriefing phase. What is equally important for any company on its way towards reuse is the communication about it. Communication among developers can be supported by including the reuse idea in basic training for beginners, in special training for advanced developers, in publishing new reuse material in the house newspaper as well as in providing an on-line catalogue of available reusable stuff (with index and query language).
Programmers often fear being deprived of their special skills as a consequence of reusing other peoples artifacts and adhering to more or less stringent standards. [5] favors an opposing view: The integration of reuse practices into the software engineering process will lead to a shift in the profile of a programmer's skills from writing lines of source code to the engineering skills of finding and integrating existing solutions as an answer to a new problem.
Table 1 lists keywords of examples in literature dealing with concepts, methods, and tools which can be used in the industrial practice. These means of reuse are arranged into the categories of the management of the SE process plus the four basic steps of development and maintenance (requirements, design, code, and test). The levels on which know-how can be reused efficiently are divided into the classes project-wide (P), branch-wide (B), and organization-wide (O). The most widespread reuse-tools are Component Libraries, object-oriented (OO) environments, and CASE-tools. Component Libraries ([2], [11], [12], [14], [16], [18], [26]) range from simple databases for pieces of code, along more elaborate systems with version management to systems especially tailored to a company's reuse procedures and needs.

	Management	Requirements	Design	Code	Test
Concept	Process Models [3] (PBO) Quality Assurance (PBO) Incentives [33], [15] (BO)	Domain Analysis [1] (Bo) Requirements Apprentice [28] (Pb)	Design Apprentice [34] (Pb)	Application Generator [7] (p) Problem Oriented Language [7] (p)	reuse of test cases (P)
Method	Software Factory [21] (PBO) Cost Estimation Models (Pb) Project Evaluation Metrics (PBo)	Draco [24] (Pb) OOA (PBo) expert advice, training of novices (PBO)	OOD (PBo) Parameter-ized Pro-gramming [17] (pBo) Generalized Component s [29] (Bo)	Coding for Portability (PB) Component Classificati on Schema [16], [27] (pBO)	Conf. Mngmnt. of test suites [6] (Pb) limiting the testobject by reusing well-tested parts (Pb)
Tool	built-in process model of a CASE-Tool (P)	general domain model in an OOA-Tool (Bo)	module templates (pBo) OO-class library (Pb) SE-environ-ment (PBo)	Component Library [13], [11] (PBo) CASE-Repository (P)	capture/ replay tool [4] (Pb)

Tab. 1: Concepts, methods and tools for reuse in practice:
The level of usual efficiency of a measure in a class is indicated by the size of the corresponding letter: impractical: no letter at all, some efficiency: small letter, high efficiency: capital letter project-wide (P), branch-wide (B), organization-wide (O)

OO environments ([20], [23]) are tempting due to their built-in inheritance/late binding/reuse possibilities. These features are a blessing in conjunction with strict adherence to common standards and design models; otherwise the programmer will be lost in a web of ill-defined objects and unclear roles of these objects in the system without any overview ([31]).
[21] considers CASE tools as the ideal basis for reuse. They provide a repository, built-in methods, guidelines, and process-models. This might be true if that particular CASE-tool was built with modern reuse techniques in mind. If that CASE-tool was designed without a clear reuse intention, all the built-in features will have a strong counter-effect.
Since the pay-off of code reuse is moderate compared to the reuse of designs and specifications, the representation of reusable designs is a strong research issue. According to [8] the problems in the reuse of design documents are: Intermingled domain-specific and implementation-specific decisions which cannot be reconsidered each on its own easily any more. Final specifications which contain too many specific details which keep the specification as a whole from being reused - although the more general parts were good candidates for an immediate reuse if the details could be stated in a fuzzy way.
Having a look at the SE history one can find an unbroken thread of implicit reuse by the migration of individual expert know-how from concepts to methods and finally into tools then. The implicit reuse of the huge amount of experience necessary to build a state-of-the-art SE environment is only rarely recognized by programmers who work with tools of that quality every day unless they have to use a less advanced tool from time to time.

3 A "Return of Investment" Reuse Model for Practitioners

After having made a decision towards a fundamental change in the hardware and the system software environment our client realized that most of the established strategies for in-house software development should be questioned and redesigned, too. Apart from a general rise in consciousness in real software engineering the basic aim was to go towards object orientation. (A comparable competitor in the same business field decided to move towards non-OO CASE-development. This will result in a completely different strategy). There was also a strong will in the medium management to enhance the reuse of software, and it was intuitively obvious to them that OO promises new possibilities of reuse but does not ensure them.
This frankness was the beginning of our consulting project. There was not much more than some imprecise wish of "reusing as many components as possible that have worked well already". For four months we did nothing but interview software engineers, programmers, project managers. The interesting problem at hand was heterogeneity

with respect to age and range of applications, to the range of development environments, to software engineering consciousness as well as to the level of knowledge among software engineers.
Software is mainly written for the developer's own use. It is vital to have a reliable support of the development environment. As a result of our inquiries we divided our task into two different projects:

- work out reasonable suggestions for reuse of existing software
- work out a plan for a reuse culture adapted to the new OO development

The answers to these consulting and research questions are outlined in the following three chapters. From the viewpoint of reuse the minimum quality level is maintainability of a project. Maintainability is a precondition for the quality level of balance, which includes a clear structure of software documents from general to specific ones within a certain domain. The most advanced level we title "standardization". Maintainability and balance are preconditions for standardization. The title was chosen because of the fact that within a SE culture of "searching, selecting and putting together" software document components obviously have to be standardized to a high degree.

3.1 Maturity level I: Maintainability = Reuse in one single Project

Large data processing companies that have been on the job for 25 years now have a wide-ranging supply of applications which often have been designed for different types of operating systems and work with different generations of hardware, too. Quite frequently working software systems are ported onto a new hardware/operating system environment with the help of emulations. If software systems have been developed very carefully, requirements in some cases still fit to the actual code even after 10 years. Others are perfectly doing important services, but are not documented at all. Most of the information has flowed informally during development and maintenance. Re-engineering was never regarded important. Developers of these programs often still work in the company and enjoy the benefit of their irreplaceable position.

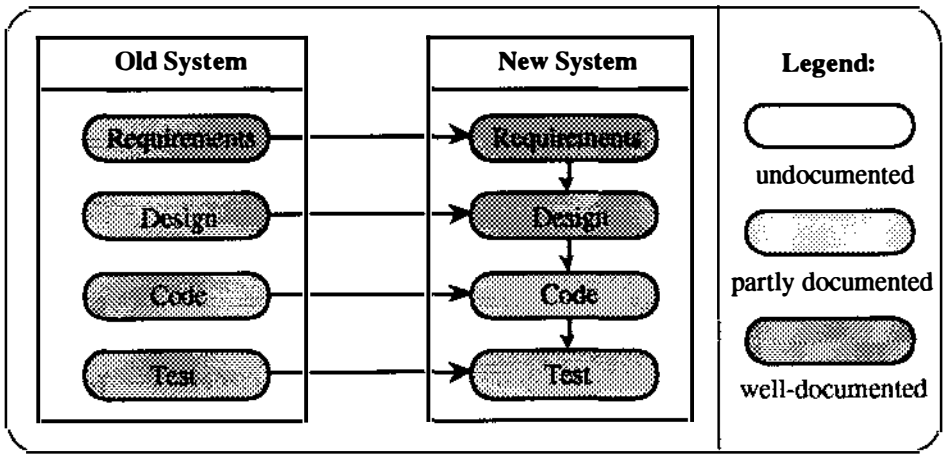


Fig. 1. Reuse level I: Conscious maintenance of a well designed software system (adapted from [3])

It is not software engineering, but it is reality. Software engineering, though, should try to have an emphasis on predictable risks rather than on adventurous travels.
Figure 1 shows the ideal reuse of requirement definitions, design documents, code and testing experience, in case some new requirement enforces a change in the old software system. In the best case change of the requirements will result in a change in the subsequent phases, while reusing the contents of the corresponding documents of the old system. In this sense good reuse within one single project is just good maintainability. [3] has pointed out this connection by "design for maintenance is design for reuse".

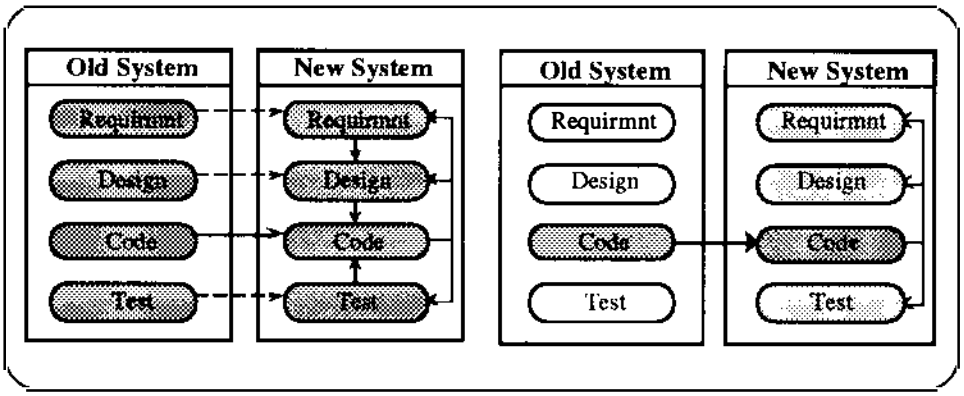


Fig.2. Reuse level I: Typical quick maintenance of a normally designed software system (left), conscious quick maintenance of an ill designed (though not untypical) software system (right)

Figure 2 shows what usually happens to a software system when being maintained. Either there is a quick fix of the code satisfying the new requirements. This allows a quick delivery of the new system. Afterwards the documents of the other phases are adapted to the changes in the code. Though this does not match good style of software engineering (fig. 1), it is rather practical and much better than the "maintenance ignorance" towards informal information flow. Moreover, from this "quick fix maintaining process" one can retrieve a cost/time efficient reuse activity for ill designed software (fig. 2, right side). Whenever changes are made, these changes are made at the code. Afterwards maintaining stuff should start a short re-engineering activity, using their momentary insight for producing documents on other phases reporting on the changes they made as well as on the insight they had to gain about the whole system. This can lead to a software system which is "a little less ill designed". (A purer solution would be, of course, to start a complete re-engineering of the software system on hand. Nevertheless, often you cannot enforce that for the reasons described before).

In the client's case especially older programs turned out to be nearly undocumented. Often requirements as well as abstract designs were missing. These software components have been developed during the last 10 years, some will be used for the next 5 to 10 years to come. They come from programmers to whom at that time anything but code is "non-productive theory". The according project managers regarded documenting as hindering countable progress. High maintenance costs had no negative effects on them.

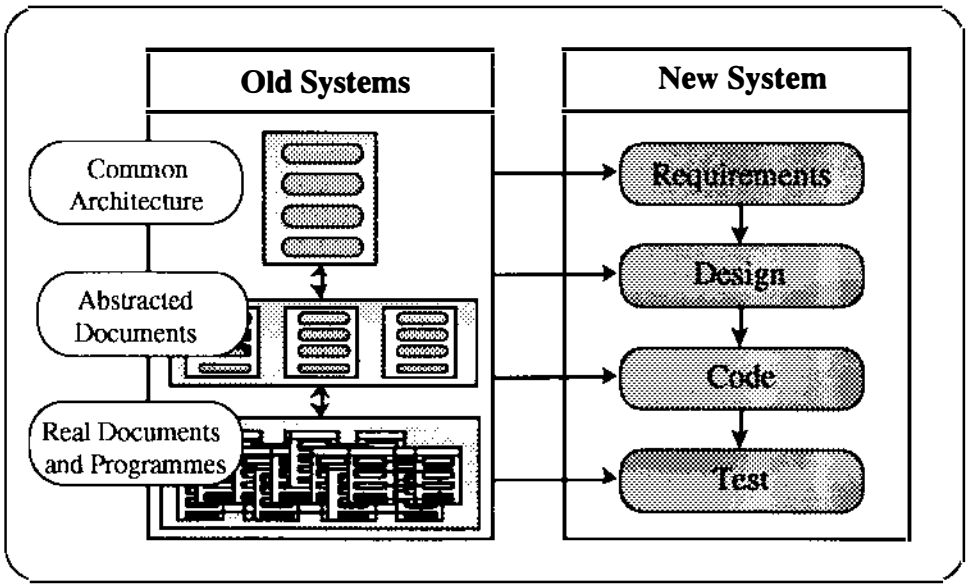


Fig. 3. Reuse level II:
Development of a software system using well structured and balanced information on requirements, design, code and test from similar projects

For software of that (non-)quality an overall redesign is much too expensive. For those pieces of software that are frequently maintained or those which are likely to be changed it is reasonable to assure maintainability.
A large number of assembler code pieces (some 2000) written between 1977 to 1985 providing application-oriented data transactions in the host environment was a typical example in the investigated environment.
In general, this maintainability level of reuse should be regarded as a minimum level. It is reasonable and appropriate also for uncommon new projects, too, where the high starting efforts for reuse are not (yet) justified.

3.2 Maturity level II: Balance = Reuse within similar Projects

Maintainability means that some software system can be modified within limited and predictable efforts. Nevertheless a maintainable system is not necessarily well prepared for a major change in requirements. Any broader range of reusability requires a more general design. Any software system contains specific and general components. More general documents are more likely to be reused in similar projects than more specific ones. We define a group of software systems as balanced, if there is a clear top-down structure from the general to the specific in documents concerning analysis, design, code and test. This balanced structure can be achieved by a consciousness during development or by redesign investment during maintenance.
There are several ways to gain the level of balance: E.g., a senior programmer who is experienced within a branch of applications can design a skeleton of documents and modules that serve as a sort of abstract language dealing with data structures, file structures and procedures for the domain at hand. Balance comes along with conscious redesign and abstraction as well as with retrieval of general components. Good (old) modular design of a software system results in a balanced structure, too. Obviously good OO analysis and OO design have to result in a balanced structure.
The client's older software systems (see 3.1) afford considerable extra efforts to gain a balanced software structure. Within our sample environment we found a group of approx. 5000 small to medium sized PL/I batch services, which might be restructured requiring an estimated investment of 6-8 person years.
What makes this level of reusability different from the following reuse culture is its mere domain-specific balance without a special design towards reuse. Standardization management is missing. In a newly developed project a balanced structure can originate from an extensive domain analysis. The level of balance ensures maintainability and time/cost reasonable reuse within similar projects.

4 Level III: Standardization = Reuse Culture in Practice

Balanced software systems are not the outcome of a specific reuse strategy, they arise from apriori consciousness in the SE process or from some extra investment into the structural design of a software system. This level II maturity of reusability is a side effect of a clear and understandable structure allowing more efficient development in similar projects.
Establishing a real reuse culture affords several technical and organizational efforts. Design of a new project goes along with the use of repositories for all phases of the development. Making a reuse culture work will need developing, providing and enforcing of standards.
The use of quality assurance methodology is required on the technical level. On the organizational level coordination of quality assurance activities (metric analysis, quality check through reviews) and reuse activities is necessary. Reuse and quality assurance must be taught to programmers. It must be re-enforced and rewarded as well. Figure 4 shows an overview of the roles involved within a reuse culture.
The following paragraphs explain the actual activities and responsibilities of the people involved within a reuse culture. The role model presented is suitable for the client's rather large software development department. For smaller companies the model can be adapted by an appropriate melting of roles (e.g. TM = MM, BL = TL, GL = PR, QA = RE = TA = RC, for explanation of the acronyms see fig. 4).
Top-Management (TM) communicates reuse as part of the company's culture, sets strategic reuse goals. TM enforces MM to start and keep up reuse research, funds reuse projects and training, and establishes RC on corporate level.
Middle-Management (MM) develops procedures and standards for respecting reuse in "everyday" project practice. MM starts and evaluates reuse test projects with help of TAs, structures the software development by reuse domains based on the stability and importance of the organization's activities in each domain, and allocates REs to project groups. MM provides incentive rewards to reuse participants, finds a solution to legal issues of potential liability and partial ownership for reuse software in contracts, establishes with the help of QA a central database for the financial data of software development, maintenance, and reuse for all projects, requires reuse within related projects.

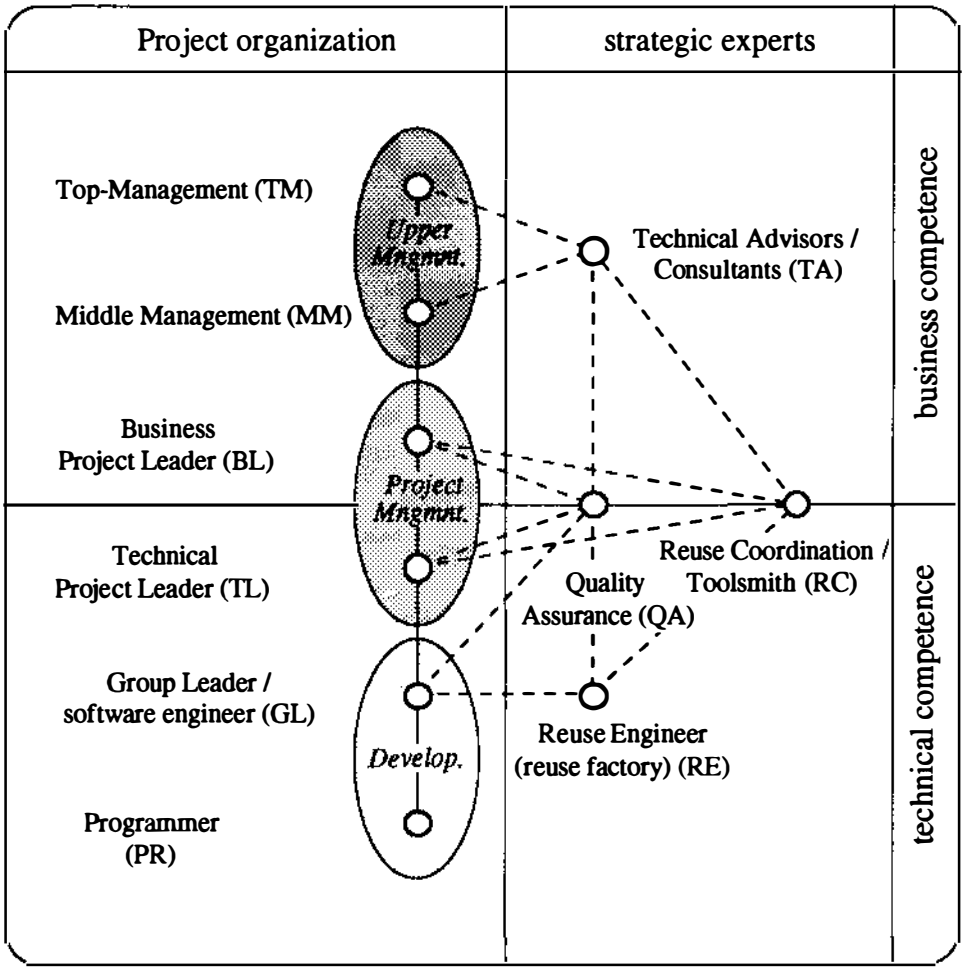


Fig. 4. A taxonomy of roles with respect to
a) project level and strategic level , b) technical and business competence

Project Management (PM) enforces reuse with all project participants.
Business Project Leader (BL) seeks contractual means to encourage contractors to create reusable software and to reuse software, tries to alter project funding approaches with help of MM to encourage the creation of reusable software.
Technical Project Leader (TL) emphasizes in his project effective and consistent methods for all aspects of software development and maintenance with GLs and PRs, establishes strong connections between future maintenance and reuse, makes use of available requirement components and takes advantage of existing high-level designs to create reusable requirements components, keeps the software reusable within the maintenance phase, states the reuse of software and the creation of reusable software as a requirement, evaluates the possible use of OO approaches or an application generator, establishes a set of organizational guidelines for code development, parameterizes specifications that are dependent on the machine environment, emphasizes the use of metrics to assess adaptation effort.
Group Leader/ Software Engineer (GL) distinguishes during design among several existing components with similar functionality, makes use of available detailed design units and takes advantage of available code modules, tries to reuse test cases for the integration test phase, constructs code for portability and adaptability.
Programmer (PR) Makes use of available code modules and test cases during coding and unit test, creates reusable code components, when coding uses generics, parameterized procedures, and code templates for greater generality, emphasizes good programming style for better understand-ability, documents each component on-line.
Technical Advisors / Consultants (TA) provide state of the art technical know how to upper management, make an assessment of current SE practices and remedy major strategic shortcomings.

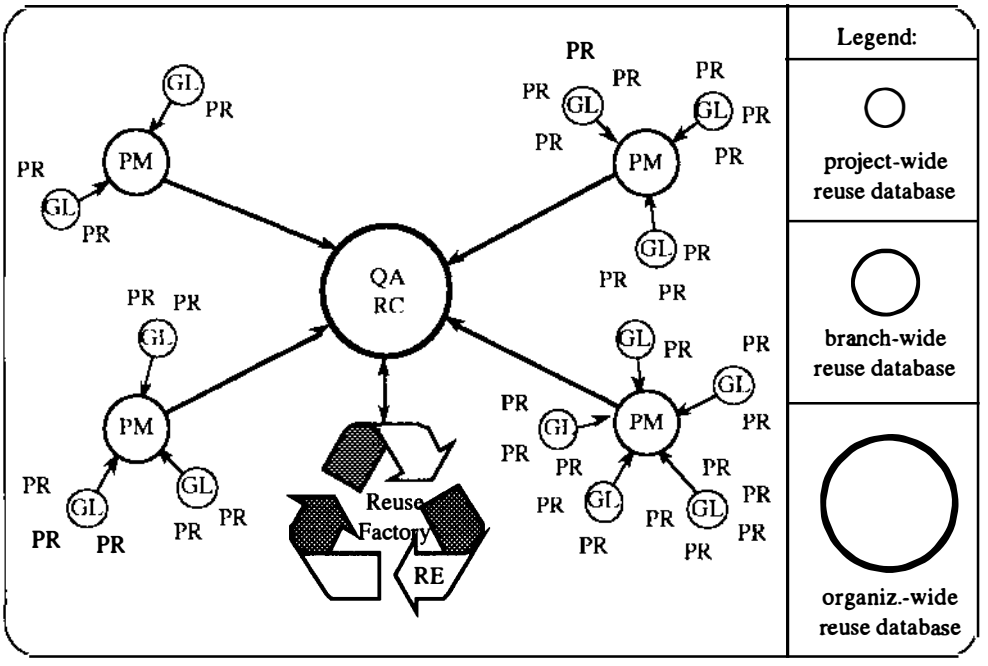


Fig. 5. Reuse level III:
Collecting and integrating information for reusable documents, code and standards
(PR = programmer, GL = group leader, PM = project manager,
QA = quality assurance, RC = reuse coordination)

Quality Assurance (QA) enforces the technical standards of MM on the developed software and development process, checks software which is to be stored in the organization-wide reuse database for fulfilling the requirements for reusable components.

Reuse Coordination (RC) uses domain analysis results as a basis for identifying reusable components, takes advantage of existing system engineering analyses, provides domain analysis results via REs to project groups, assesses to decide whether the development of a reusable software component by the reuse factory is advisable, sets standards to be met by all central library components, determines approaches for classifying and storing components, develops and implements a mechanism for search and retrieval in the database with a user-friendly interface, gets feedback from users of components.

Reuse Engineers (RE) REs in the Reuse Factory are responsible for the creation and maintenance of components in the organization-wide database. Moreover they act as advisers for individual projects to keep up information flow between the reuse-oriented developing staff an project-developers, record and supply adaptation suggestions with a reusable component.

The model in figure 5 assumes three levels of reuse databases: Within a project the GLs identify those pieces of software which are of use for the participants in the same project. These components are stored and communicated via the project-wide database. A PM can promote software from this database to the branch-wide database. The branch-wide databases are used by PMs from similar projects and are controlled by the RC. The RC looks for components of general interest which are worth 'polishing' for organization-wide reuse. These components have to correspond to reuse standards. These standards are maintained by the QA who reviews all components which are suggested for incorporation into the organization-wide database. If there are violations of standards the component is handed to the reuse factory for upgrading.

The purpose of this hierarchic promotion of reusable components is to assure that a component require the appropriate reuse investment (according to its distribution scope). Only few components migrate to the organization-wide database thus needing closer observance of QA, RC, and REs.

Software engineers reuse elements from local, branch-wide, and organization databases. A more local component is closer to the people who created it. The more widely available a component is, the more general, well tested, and standardized it is.

This three-level database model is a practical compromise between the common 'put all components into one big pot' reuse approach (which leads to indigestible hot-pot) and the common "drawer-reuse" in which every programmer and designer reuses software from previous personal only or 'trusted' experience.

Currently the client is implementing the new software developing strategy as well as most of the ideas presented in this paper. Two level II projects have been started. Stuff has been hired as well as in-house developers have been trained to cover the roles of a reuse culture. Several prototypes of applications are developed. By now we are not able to provide the reader with systematic results.

5 Conclusions

Within this article we reported on a software reuse model for a rather heterogeneous software development environment. We identified three levels of maturity for reuse in practice. These levels have been deduced from a detailed study of theoretical research contributions as well as from the organizational, political, technical and educational situation we identified during our consulting interviews. Beyond these results that are valid to a very local extent only we want to generalize our experiences by the following theses:

- A simple reuse activity - which should be and can be applied whenever an old piece of software is changed - is (re)establishing maintainability.
- Projects which are similar to each other, systems within a homogeneous domain, are well suited for redesign extracting a "balanced hierarchy" of software documents. Balance results in a sort of domain language providing working components on different levels of abstraction. Nevertheless this reuse effort is done without the cost of explicit quality assurance and reuse management.
- Strategic reuse efforts - the most mature approach - afford a general technical management of these efforts which can be established in a cost/time efficient way for newly developed software only. It will demand fixed investments in quality assurance, standardization, and reuse techniques. Moreover, a completely different style of programming is required from every software engineer for the development of software.

A working reuse culture can be established and maintained most easily with the help of good tools. Nevertheless, CASE-tools today do not provide satisfying reuse facilities. So far, the only way to succeed seems to concentrate the task force of your most experienced as well as your most innovative software engineers; make them define and standardize an environment big enough to serve as a beginning. Make them develop and guide the development of several "real" projects and adapt your environment permanently. Finally, when you start spreading "the new religion" within your company do not forget to integrate persons who were in important positions before the changes and adapt formerly significant standards to the new tasks.
Be aware that nobody might use your environment, unless you award

- coding by finding (in the reuse databases)
- coding and delivering (to the reuse databases).

There is still a long way to go in practice from the conventional development approach for an application to the "discovery" of an application within a reuse environment.

References

1. G. Arango: Domain Analysis: From art form to Engineering discipline. ACM, pp.152 - 159 (1989)
2. S.P., Arnold, S.L. Stepoway: The Reuse System: Cataloging and Retrieval of Reuse Software. Proceedings of COMPCON 1987, pp. 376-379
3. V.R. Basili, H.D. Rombach: Support for comprehensive reuse,. Software Eng. J., Sept. 1991, pp. 303 - 316
4. B. Beizer: Software Testing Techniques. Van Nostrand Reinhold, 2nd ed., (1990)
5. L.A. Belady: Foreword. in: Software Reusability (vol. I), Concepts and Models,. Addison Wesley, p. vii-viii (1989)
6. H. Berlack: Software Configuration Management. Wiley 1992
7. T.J. Biggerstaff, A.J. Perlis: Foreword (Special issue on software Reusability). IEEE Transactions on SE, Sept. 1984, pp. 474 - 476
8. T. Biggerstaff, Ch. Richter: Reusability Framework, Assessment, and Directions. IEEE Software 4(2): 41-49, March 1987
9. T. Biggerstaff: Design recovery for maintenance and reuse. IEEE Computer 22(7): 36-49 (1989)
10. T.J. Biggerstaff, A.J. Perlis: Software Reusability. ACM-Press 1989
11. B.A. Burton, R.W. Aragon: The reusable software Library. IEEE Software, July 1987, pp. 25-33
12. G. Caldiera, V. Basili: Identifying and Qualifying Reusable software Components. IEEE Computer, Feb. 91, pp. 61-70

13. M. Cavaliere: Reusable Code at the Hartford Insurance Group. in: T. Biggerstaff and A. Perlis: "Software Reusability - Vol. 2, Applications and Experience". Addison Wesley and acm Press, 1989, and in: Proc. of the ITT Workshop on Reusability in Programming, Newport R. I., 1983,
14. P. Devanbu, R.J. Brachman: LaSSIE: A Knowledge-Based software Information System. CACM 34(5): 34-49, May 1991
15. R. Fairley, S. Pfleeger et al.: Final Report: Incentives for Reuse of Ada Components. vols. 1 through 5, George Mason University, Fairfax, Va., 1989
16. W.B. Frakes, B.A. Nejimeh: An Information System for Software Reuse. Proceedings of the Tenth Minnowbrook Workshop on Software Reuse 1987
17. J. Goguen: Principles of Parameterized Programming. in: T. Biggerstaff and A. Perlis: Software Reusability - Vol. 1, Concepts and Models. Addison Wesley and acm Press, 1989 (ext. vers. of IEEE TR-SE, Sept 84)
18. G.E. Kaiser, D. Garlan: Melding Software Systems from Reusable Building Blocks. IEEE Software, July 1987, pp. 17 - 24
19. R.G. Lanergan, Ch.A. Grasso: Software Engineering with Reusable Designs and Code. IEEE TSE SE-10(5): 498-501, Sept. 1984
20. J.A. Lewis, S.M. Henry, D.G. Kafura: An Empirical Study of the Object-Oriented Paradigma and Software Reuse. ACM Sigplan Notices, OOPSLA '91, p.184-196
21. Y. Matsumoto: A Software Factory: An Overall Approach to Software Production. Proc. of the ITT Workshop on Reusability in Programming, Newport R.I., 1983 and in FREEMAN P.: Tutorial "Software Reusability"; IEEE-CS Press, 1987
22. B. Meyer: Reusability: The Case of Object-Oriented Design. IEEE Software 4, 2: 50-64, March 1987
23. W. Myers: We Want to Write Less Code. Computer 23(7): 117-118, July 1990
24. J. Neighbors: The Draco Approach to Constructing software from Reusable Components. IEEE TSE, SE-10(5), p. 564 - 574, Sept. 1984
25. R. Prieto-Diaz, P. Freeman: Classifying Software For Reusability, IEEE Software 4, 1: 6-16, 1987
26. R. Prieto-Diaz: Domain analysis for reusability, Proc. of COMPSAC 87, Tokyo, Japan,. pp. 23-29
27. R. Prieto-Diaz: Making software Reuse work: an implementation model. ACM SIGSOFT SE Not. 16(3): 61-68, July 1991
28. H.B. Reubenstein, R.C. Waters: The Requirements Apprentice: Automated Assistance for Requirements Acquisition. IEEE TSE, Vol. 17, No.3, March 91, pp. 226-240
29. R. Riehle: Software components - designing generalized components by creating abstractions. Programmers J., pp. 75-78, Nov./Dec. 1990
30. R. Selby: Quantitative Studies of Software Reuse. in: Software Reusability: Vol. II Applications and Experience, ed. T. Biggerstaff and A.J.Perlis, 213-33, 1989
31. D. Taenzer, M. Ganti, S. Podar: Object-Oriented Software Reuse: The Yoyo Problem. Journal of Object-Oriented Programming 2(3):30-35, 1989
32. W. Tracz: Reusability Comes on Age. IEEE Software, July 1987, pp. 6-8
33. W. Tracz: Software Reuse: Motivators and Inhibitors. Proceedings of COMPCON S 87, pp. 358-363
34. R.C. Waters, Y.M. Tan: Toward a Design Apprentice: Supporting Reuse and Evolution in Software Design. Software Engineering Notes 16(2): 33-44, 1991

Test Suites for C++ Class Libraries

Daniel Hoffman

University of Victoria

Department of Computer Science

P.O. Box 3055

Victoria, B.C., Canada V8W 3P6

dhoffman@csr.uvic.ca

Abstract

In contrast to the explosion of activity in object-oriented design and programming, little attention has been given to object testing. This talk will describe techniques for automated testing of C++ class libraries.

Biography

Dr. Daniel Hoffman received the B.A. degree in mathematics from the State University of New York, Binghamton, in 1974, and the M.S. and Ph.D. degrees in computer science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial programmer/analyst. In 1992, he was on leave at Tandem Computers in Cupertino, California. He is currently an Associate Professor of Computer Science at the University of Victoria. His research area is software engineering, emphasizing the industrial application of software specification, verification and testing.

Test Suites for C++ Class Libraries

Dan Hoffman
Department of Computer Science
University of Victoria
British Columbia, Canada

Talk Overview

- Class libraries
  - what are they?
  - why is class testing important?
- The testgraph methodology
  - a new kind of graphical programming
- Case study
  - methodology applied to commercial code

Class Library Overview

- Graphical user interface (GUI) libraries
  - window and menu creation, manipulation
  - keyboard, mouse event handling
  - interacts closely with environment

- Collection (or container) class libraries
  - object-oriented abstract data types
  - queue, list, set, tree, sort, etc.
  - interaction entirely through calls, return values

Class Testing

- Reusability is the cornerstone of the OO dream
  - class libraries a major source of reuse

- Class library reliability is critical
  - library user will not tolerate frequent bugs

- Thorough testing essential
  - else the object-oriented dream will be a nightmare

- Status: testing tools and techniques
  - very little work in university or industry
  - object-oriented software engineering in jeopardy

The Class IntSet

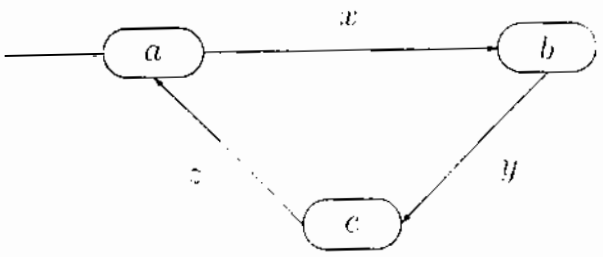
```
const int MAXSIZ 10;
class IntSet {
public:      IntSet();
        void add(int);       //duplicate, full
        void remove(int);    //notfound
        int  is_member(int);
}
```

- ◯ Assume: `IntSet s;`
- ◯ After `s.add(25)` `s.is_member(25)` returns 1
- ◯ After `s.add(25); s.remove(25); s.is_member(25)` returns 0
- ◯ After `s.add(25); s.add(25);` throws exception `duplicate`

IntSet Test Requirements

- ◯ Object states
  - set size: 0, 1, `MAXSIZ`
- ◯ Operations: normal case
  - `add`, `remove`, `is_member`
  - on first, last, middle elements
- ◯ Operations: Exceptions
  - `duplicate`, `notfound`, `full`
- ◯ Operations must be applied to each object state

Testgraphs

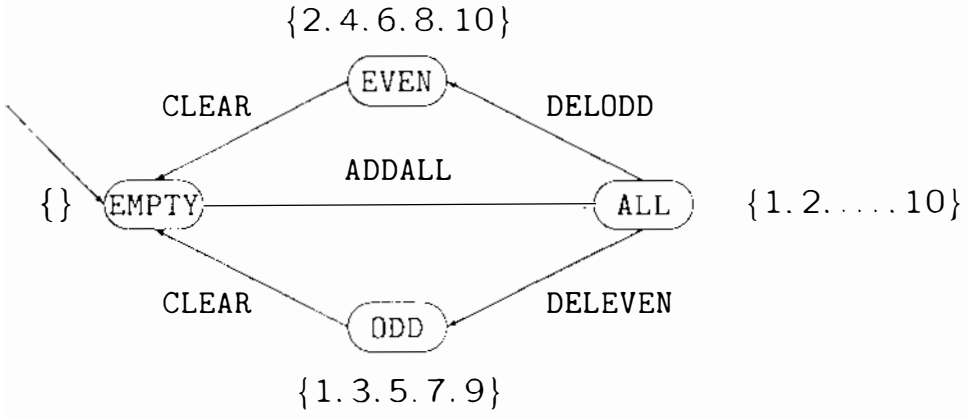


- Path, rooted path
- Coverage: statement, arc, all-paths
- Arc coverage
  - NO: $\langle a, b, c \rangle$
  - YES: $\langle a, b, c, a \rangle$

Testgraphs—Two Views

- CUT: Class-Under-Test
- State-machine interpretation
  - node: CUT state
  - root node: initial CUT state
  - arc: CUT state transition
- Operational interpretation
  - driver functions `arc(int)`, `nod`
  - `nod` called each time a nod is reached
  - `arc(`x`)` called each time an arc with label x is traversed
  - meaning of a path: the associated `nod`/`arc` call sequence

IntSet Testgraph



○ {⟨EMPTY. ALL. EVEN. EMPTY⟩. ⟨EMPTY. ALL. ODD. EMPTY⟩}

```
nod(); arc(ADDALL); nod(); arc(DELODD);
      nod(); arc(CLEAR); nod();
```

Oracle Strategy

○ Test oracle

- mechanism for determining output correctness

○ Our oracles

- one oracle class for each CUT
- provide roughly the same operations as the CUT
- support only the testgraph states and transitions

Class IntSet0

```
typedef enum EMPTY, ODD, EVEN, ALL nodid;
class IntSet0 {
public:       IntSet0(nodid,int);
        void  add(nodid);
        void  remove(nodid);
        int   is_member(int);
        nodid n;                    // node identifier
        int   siz;                  // graph parameter }
```

- Bit representation of nodid

  EMPTY: 00 ODD: 01 EVEN:10 ALL: 11

- Assume: IntSet0 so;
- After so.add(EVEN) so.is_member(2) returns 1
- After so.add(EVEN) so.add(EVEN) does nothing

IntSet0 Implementation

```
int IntSet0::is_member(int x)
{
      switch (nodid) {
      case EMPTY: return(0);
      case ODD:   return(x%2 != 0);
      case EVEN:  return(x%2 == 0);
      case ALL:   return(1); }
}

IntSet0& IntSet0::add(nodid n1)
{
      n = n | n1;
}
```

Utility function—loadIntSet

```
// invoke s1.add(i) for every i in n1
void IntSetD::loadIntSet(IntSet& s1, nodid n1, int siz)
{     IntSetO *so1;

      so1 = new IntSetO(n1, siz);
      for (int i = 1; i <= siz; i++)
            if (so1.is_member(i))
                  s1.add(i);
      delete so1;
}
```

arc Implementation

○ Assume: IntSetD declares IntSet s; and IntSetO(EMPTY,10) so;

```
void IntSetD::arc(int a)
{
      switch (a) {
      case ADDALL:  so.add(ALL);
                    loadIntSet(s, ALL, so.siz);
                    break;
      case DELODD:  ...
      case DELEVEN: ...
      case CLEAR:   ...  }
      delete so1;
}
```

nod Implementation

◯ Assume: `IntSetD` declares `IntSet s;` and `IntSetO(EMPTY,10) so;`

```
void IntSetD::nod()
{     // normal case
      for (int i = 1; i <= so.siz; i++)
            if (s.is_member(i) != so.is_member(i))
                  // take error action

      // exceptions
      // ...
}
```

The C++ Booch Components

◯ Sixteen classes with variations
- standard ADTs, with generic element type
- bounded or unbounded; single or multi-threaded

◯ The `Set` class
- seventeen member functions, including:
  union, intersection, difference, subset
  assignment, comparison
- `s.set_union(s1)` replaces `s` with $s \cup s1$

◯ Key differences with respect to testing
- many more member functions
- operations on pairs of sets
- generic element type

nod Pseudocode

- Assume: `IntSet s; IntSetO so;`

- Normal case

  check that `s` and `so` have same elements

  for each of the nine `IntSet` operations on pairs of sets

  for each of the sets EMPTY, ODD, EVEN, ALL

  check that `s`, `so` have same behavior

Results

Class name	# member functions	source file size in lines declaration	implementation
`SetO`	16	49	138
`SetD`	11	21	240
`ExcTbl`	9	25	84
`TestGraph`	19	56	321

- No normal case failures
  - `Set` implementation simple

- Several exception failures
  - exception signaled correctly
  - object state changed anyway
  - `add(1); add(1);` adds 1 twice

Conclusions

- The importance of class testing
  - reuse depends on reliability
  - reliability depends on thorough testing

- Effective testing of collection classes
  - will exploit the call-based interface
  - *not* keystroke/mouse-click capture/playback
  - *not* screen/file capture/comparison

- The testgraph methodology
  - testgraphs: abstraction of the CUT state machine
  - oracles: provide CUT operations on testgraph states
  - thorough testing possible with simple testgraphs, oracles

Building Extensible Object-Oriented Systems

Jim Donahue
Documentum, Inc.

Abstract

One of the claimed virtues of object orientation is extensibility; unfortunately, many object-oriented programs aren't very extensible. Techniques that improve the extensibility of object-oriented systems will be discussed

Presenter

Jim Donahue, is a Senior Engineer at Documentum, where he has been architecting the infrastructure of an object-oriented document-management system. Previously, he directed the Olivetti Research Center and did research on programming languages and database systems at Xerox PARC. He is one of the designers of the Modula-3 programming language.

Building Extensible Object-Oriented Systems

- Extensibility -- What?
- Extensibility -- Why?
- Extensibility and object orientation
- Hints for writing extensible systems

Disclaimer

- This talk is based entirely on experiences at Documentum.
- The views expressed are the author's and do not necessarily reflect those of the Documentum management.
- Your mileage will be lower. Void in Nebraska.

Extensibility: What?

- Oldest module continues to get older
  Code can be written and ignored
- Only constant number of modules affected by new features
  New features do not cause widespread tweaking
- Modules have similar structure
  New features added by following basic recipes

Extensibility: Why?

Imperatives of Software Development

- As quickly as possible ...
- As fully featured as possible ...
- As little support as possible ...

Extensibility: Why?

- Old code gets older
  Add more functionality quicker by not doing rewrites
  Code that works continues to work
- Only constant amount of code is changed for new features
  Code that works continues to work
  Can continue to add new functionality to meet market needs
- Code has common structure
  Less thought has to go into coding of next feature
  Commonly used structures tend to have fewer bugs

Object Orientation and Extensibility

- Objects = Interface + Inheritance + Overloading
- Interfaces provide firewalls
- Inheritance allows reuse of working implementations

- Overloading allows reuse of working clients

What the C++ Books Don't Tell You

- Not everything should be an object
  Interfaces of procedures have their place
  This seems very hard for Smalltalk programmers to grasp
  "Procedure-oriented and object-oriented views of programming are fundamentally different" -- BS

- There is a logical notion of a "complete" set of methods
  More than providing "get" and "set" methods for private members
  Describing objects in "real world" terms doesn't help

- Use inheritance to break large tasks into small pieces
  Chose simple themes for classes
  It's OK to use a class entirely for internal purposes

More Things the C++ Books Don't Say

- Function parameters make *lots* of things simpler
  Are "friends" always necessary?
  None of the C++ books show *any* examples
  Better still -- abstract "closure" objects

- Abstract classes essential to making system extensible
  Case statement may be symptom of bad design
  Only "fully abstract" classes are closures

- "Void *" still has its uses
  If you don't have templates, better than macros
  Sometimes only thing you can say for function parameters, e.g.,

```
void delete_proc( void *object )
```

Beyond C++: Hints for Extensible Design

- Self-description

- Self-healing

- Extension through procedure registration

- When extension fails, make sure it breaks

Self-Description

- Basic Law of Extensible Systems: Things change

  Problem is how to write code that doesn't change when the data it operates on does

  Write code that you can lock in a vault

- If you make the data self-describing, you can write "persistent code" that will survive changes in data representations

- Example:

  Files that describe server options (where files are saved, what conversions are supported, how documents are printed or mailed, ...)

  How to write program that queries these files?

  Answer: Build self-description interface to files

Self-Healing

- Change introduce inconsistencies

  Little changes in data representation require modifications of existing files, databases

  Separate conversion programs are problematical

- Some inconsistencies can be handled "automagically"

  Need to have standard way to mark "this is broken"

  Need to have standard way of performing fix

- Version stamp the data -- change stamp with data representation change

- Check stamps during startup, performing modifications as necessary

Procedure Registrations

- A simple technique that isn't used widely

- Provide procedures for clients to "register" callbacks

- Commonly used for "event" processing
  Window managers
  But lots of other valuable uses ...

- Mapping external names into objects of various subtypes
  Put "tag" in name
  Register "create" procs for each tag

- Other type-specific behavior
  E.g., procedures to perform special actions for object dumping/loading

When Extension Fails, Make Sure It Breaks

- Using compiler as debugging tool hard for C programmers to grasp

- Try to capture design decisions in interfaces
  Combination of argument types, object methods, constant definitions

- Make implementations totally dependent on interfaces
  Changes in interfaces mean implementations either:
    Compile and work
    Don't compile

- New skill is managing interfaces:
  Avoiding gratuitous dependencies
  Innocuous changes can cause everything to be recompiled

Conclusions

- Designing for extensibility requires asking simple question:
  "How can I write this code just once?"

- Objects help extensibility, but are neither necessary nor sufficient

- Lots of object-oriented texts miss the points:
  Extensible programs mix of objects and procedures
  Self-description, self-healing orthogonal to objects
  Need to take advantage of, manage interfaces

Improving Software Quality Through Practical CMM Level Progression

by

Larry Cousin
3M Health Information Systems
575 W. Murray Blvd.
P.O. Box 57900
Murray, UT 84157-0900
(801) 265-4565
larryc@code3.com

Abstract

The last two decades have seen a significant rise in the awareness that product quality is essential to the viability of a producing organization. Nowhere has this been more true than in the software industry. Software production organizations that do not produce products of high operational and functional quality in a timely manner do not remain in existence. This is the reason that the Software Engineering Institute (SEI) developed a model called the Capability Maturity Model (CMM) and philosophy for its use in aiding an organization to raise itself to a higher level of ability in producing quality software on-time and within budget. One potential shortcoming with the model is, however, that an organization is expected to explicitly follow SEI's prescribed program for progress to occur. This paper presents a practical program for the use of CMM principles by organizations which cannot conform exactly to CMM procedure. This program, called the eclectic model of CMM application, has been employed at 3M Health Information Systems. Initial results indicate that software quality and productivity can be improved by employing this modified version of CMM.

Larry Cousin received his Ph.D. degree in Computer Science from Arizona State University in 1989. He is currently employed as a Senior Software Engineer for 3M Health Information Systems doing research in advanced clinical fourth generation languages, analyzing the use of neural networks and fuzzy logic in clinical data analysis, and working on a team which has the purpose of improving software quality through integration of CMM process principles. He has worked for over thirteen years doing software engineering and research for several large corporations. His research interests include software engineering, software maintenance, language theory and design, data bases, neural networks, and artificial intelligence.

1. Introduction

In the keynote address of the 1992 Conference on Software Maintenance, Bill Curtis of the Software Engineering Institute made the following timely observation [Curtis92]:

> "The software engineering community is learning, as has virtually every other area of engineering, that advances in productivity and quality do not materialize just because technology was thrown at a problem. Rather, improving the results of software development or maintenance operations requires attention to the processes by which these operations are performed."

This attitude of pursuing improved productivity and quality by emphasizing the process has of late been recognized as essential in the Software Engineering community [Humphrey91, Osterweil87]. This recognition has been greatly influenced by the "quality movement" which has swept the United States and many other parts of the world during the last two decades [Crosby79, Deming82].

The Software Engineering Institute (SEI) is one of the major forces behind the infusion into corporate America of software process awareness and action for process improvement. SEI's Capability Maturity Model (CMM) [Paulk91, Weber91] provides software production organizations with a useful instrument to assess their level of software engineering sophistication. Also, the model provides a map of sorts which, if followed, can lead an organization towards greater and greater levels of software development sophistication. The premise is that by progressing in these levels of sophistication, greater degrees of quality and productivity will be achieved.

This paper outlines one company's effort to pragmatically integrate the principles of CMM into its software production processes and thereby increase its level of software quality and production productivity. As a prefacing axiom to this work, we accept the principles, program, and philosophy of SEI's CMM in its entirety. We believe that by fully following the SEI's program for CMM level improvement, substantial gains in quality and productivity can be achieved and production costs will ultimately be reduced.

The problem, however, with implementing any program in the real world is resource constraints (particularly with small organizations such as ours) which restrict an organization from being able to conform to the CMM program's prescribed application. Also, an organization's corporate culture may be structured such that the application of the program would be difficult or disruptive. This is the current situation in our department of 3M. It was impossible to implement CMM as prescribed by SEI; however, we wanted to implement as much as we could because we believe in the potential gains that could be obtained by progressing through the levels of the model.

To deal with this problem, we hypothesized that similar levels of CMM progression could be effectively achieved by repeatedly applying a program of practical CMM improvement. This practical improvement consists of studying the prescribed actions of the CMM, particularly the key process areas for each level, implementing these actions from each CMM level for which we have adequate resources, showing management the positive results, and repeating this process again and again. By doing this we believe we are lifting and will continue to lift our software

production organization through the levels of CMM. This process may not be done as quickly nor as efficiently as if CMM had been strictly followed, but it is being done using the resources we have available and within the bounds of our own corporate culture.

Section 2 of this paper provides a brief discussion of CMM principles. This is given to help those readers who are not familiar with CMM to gain a basic understanding of the model as well as to lay a foundation for understanding how our application of the model's principles differs from that prescribed by SEI.

Section 3 introduces what we refer to as the eclectic model of CMM level progression. This model defines our philosophy concerning the practical application of CMM principles.

Section 4 discusses the application of the model at 3M Health Information Systems. We discuss how the eclectic model has been applied to date and give several examples of our plans to apply future iterations of the model. Also, we discuss several problems encountered in applying the model as well as what we feel we have achieved from its application.

Finally, Section 5 summarizes the results we have obtained.

2. CMM Brief

The Capability Maturity Model [Paulk91, Weber91], or CMM, consists of a high-level hierarchical model of operational software production and a philosophy for its use. The model is operational in the sense that it defines the processes which must be consistently instituted by a software production organization in order to be classified as attaining a level of capability in producing quality software in a timely manner.

The philosophy for using the model includes many different concepts, but the fundamental concept is that by progressing from lower to higher levels in the model, the quality and productivity of software production will go up (and hopefully the costs of that production will go down although this is not guaranteed by CMM).

The model itself is composed of five progressively higher levels of software production sophistication or maturity. These include:

1. Level 1: Initial / Chaotic,
2. Level 2: Repeatable,
3. Level 3: Defined,
4. Level 4: Managed,
5. Level 5: Optimizing.

Implicit to the model is the idea that a software production organization will wish to progress up the hierarchy. To do this, an organization must determine its current level. To aid in making this determination, SEI has developed an assessment process which can either be applied by an outside group (CMM consultant) or can be applied by the software organization itself (self-assessment). The result of this process is the current CMM level classification of the software organization.

Once an organization has determined where it falls in the spectrum of software production capability, it can then begin to implement the key processes which pertain to the next higher level of the model. Once that level is achieved, the organization works toward the next, etc.

In this way, an organization can climb the ladder of software maturity.

One should understand that the model does not allow for an organization to reside somewhere between two or more levels. Also, to be able to progress through the hierarchy, one must step through each subsequent level. Thus, levels cannot be skipped, since by definition to progress to a higher level of sophistication, an organization must already possess (have implemented) the practices of the preceding levels.

The following defines the basic attributes for each of the five levels of the model [Paulk91]:

> "1) Initial: The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
>
> 2) Repeatable: Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar application.
>
> 3) Defined: The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software.
>
> 4) Managed: Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures.
>
> 5) Optimizing: Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies."

Each of the levels of the model is achieved by adopting into an organization the key processes of each level [Paulk91] (see Section 4) as well as the detailed practices [Weber91] which define the specifics of each process.

3. An Eclectic Model of Practical CMM Level Progression

Figure 1 gives a diagrammatic representation of the eclectic model of CMM level progression. This model is somewhat reminiscent of Boehm's spiral model of software development [Boehm88]. In Boehm's model, software is progressively refined by specifying some requirements, doing some prototyping, and performing some design and development. This work is then evaluated and amendments are made as the preceding sequence of activities is done again and again to spiral the product towards a solution to the real needs of its users. Our model of practical CMM level attainment is similar to Boehm's spiral model in that we progressively refine the processes of each level (by implementing more and more key practices) as we strive to achieve optimal ability to produce software.

As can be seen from Figure 1, our model of CMM application does not spiral per se but is more like the ringed structure of a tree. Progressing towards a level five, optimizing software production organization, is akin to having the tree be simultaneously cut from five different tangents. The cuts correspond to performing some subset of the practices related to each level.

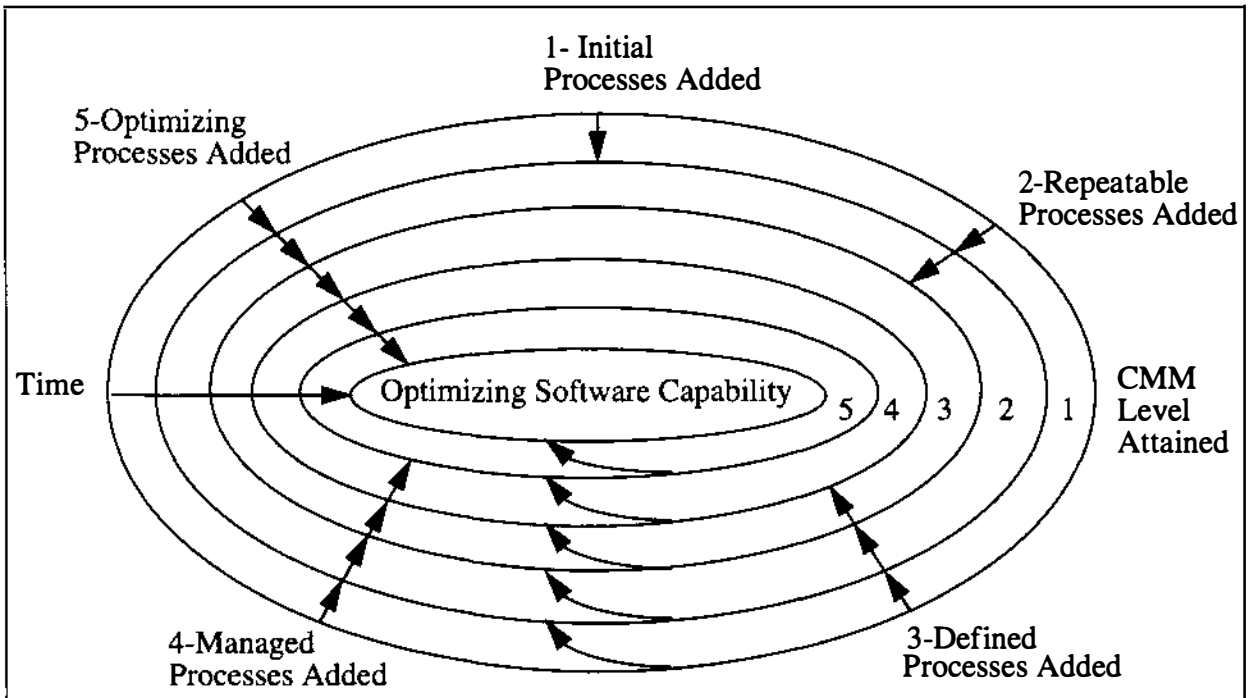


Figure 1. Eclectic Model of Practical CMM Level Progression

When all the processes have been fully adopted (an outer ring removed) for a level, we can say that the next corresponding level has been achieved. At the same time, however, we may have integrated a significant amount of key processes related to deeper levels (without fully achieving that level).

We make no assumptions as to the current level of a software organization. Of course it is important to know this at discrete points of a project to assess how much improvement has been achieved by an organization through CMM process implementation. It is not necessary in order for improvement to occur, however.

One employs the model by learning and understanding the key processes and key practices of all the levels of the CMM, evaluating within one's own organization what practically can be implemented from the processes and practices, and then implementing all of these things simultaneously. If possible, one should implement processes and practices (if they are not already being done) from the lowest levels of the model on up. The key, however, is to implement as much as possible (which may not include a full process or practice) from each of the levels at the same time. At that point one should continue to strive to fully apply and institutionalize the processes from all the levels which are being implemented.

Our belief is that by repeatedly implementing allowable processes simultaneously from each level, these processes will act as forces which will push or influence the organization towards actually attaining higher CMM levels. Of course to positively determine how capable an organization has become, CMM assessments must be performed. Our experience has shown that even if resources are not initially available to perform assessments, if the above model is applied, productivity and quality will increase and this will please management to the point that resources

will be made available to perform standard CMM activities such as the assessments.

There are several reasons why one would wish to employ the model shown in Figure 1. First, money, personnel, and other resources are not always available to adopt CMM in the manner that SEI has prescribed. Especially in these times of belt tightening and budget cuts, few organizations have the luxury to fully perform any software production activity. Thus, this model allows us to be practical in the processes which we employ.

Also, many if not most software organizations have already implemented some aspects from each of the key process areas of CMM. The model allows these organizations to continue to improve upon and maximize the processes they already have in place while slowly folding in new processes and practices.

This model is also somewhat more gentle with an existing corporate culture for an organization than pure CMM application. This corporate culture may include management individuality and autonomy as well as engineer working procedures which do not easily change. If either of these delicate environments are too quickly or too drastically mutated, any gains from improved process may be overturned by confusion, discouragement, or revolt.

Also, we don't believe that our model of process application is really that divergent from the spirit of CMM, regardless of the reason for its application. The CMM's basic philosophy is founded around disciplined application of process control. This is especially evident in level five which prescribes and requires continuous improvement. Thus, we believe the message of CMM is "do something to improve the process and don't quit." That is certainly the intent of our model, i.e., implement the CMM processes which are practical for one's own organization. Also, strive to add to these processes where possible and eventually a steady state will occur and an organization's software maturity will converge toward a higher CMM level.

We believe there comes a time for every conscientious software organization when that organization's personnel will realize that increased profit and market viability can only be achieved through process control. This is a critical mass of sorts for software--the organization will have to either change and improve its processes to meet its environment or be swept aside.

There are two keys to making our model work given the above:

1. real management support, and
2. sticking with the process to do real work with it.

The only way that either CMM or our application of it will work is to have management achieve the critical mass realization spoken of above, support the effort, and continue to support the effort. That support may not be able to initiate a full CMM program, but it must support some part of it. Then production personnel must actually use the process, not trying to work around it, to do real work. Also, they must not give up on process improvement when things go poorly for a project. From this, improvement will come and with that improvement continued corporate support to implement more process control.

4. 3M CMM Implementation

This section of the paper discusses the first iterations of 3M Health Information System's implementation of pragmatic CMM. 3M Health Information Systems is a provider of clinical information systems--software being an integral part of those systems. Our company is following the model of application shown in Figure 1. As such, components from each of the five

levels of capability are being implemented simultaneously. As productivity and quality increase and costs are reduced, we plan continued integration of missing components of CMM to gradually move the organization up through the CMM levels.

Figure 2 [Paulk91] shows the key process areas which have been defined in CMM for each maturity level. Embodied within these key processes are the key practices [Weber91] of CMM. A detailed discussion of these practices goes beyond the scope of this paper, however, suffice it to say that they define in detail what actions must be performed (but not how) to implement the processes. The sections that follow report the process components which 3M has implemented as a first iteration of our application model. These process actions are defined relative to each of the key process areas shown in Figure 2.

4.1 Level 1: Initial

The most important component of CMM at the initiation of process control is the performance of a maturity assessment. As stated above, we do not feel that this is absolutely necessary for improvement to occur if our eclectic model of use is employed. This is because we assume we are working at a level one, and in any case, the model as defined does not require an organization to know its level for CMM process application to begin. If at some point an organization wishes to know how much improvement has taken place, then assessments must be performed.

At 3M we have not performed an assessment of our capability. The reasons for this go beyond the scope of this paper. There are, however, important reasons to perform an assessment other than just to determine at what level an organization performs. The CMM documentation does not explicitly bring out these reasons, but they are none the less of paramount importance.

First of all, doing a formal assessment greatly helps middle and low-level management to be trained in the principles of CMM and process improvement. It helps to not only train them in process improvement but also to convince them of the necessity for this effort and leads to buy-in for process improvement. If they are sufficiently well trained and gain a personal conviction for the power of these principles and have the backing of high-level management, there is a greater chance that they will follow the principles. If process improvement does not have the active support of all levels of management, no program for improvement will work. On the other hand if management zeal is too great, it may smother the engineers' ability to realistically adopt the process in a timely fashion.

Doing the assessment also helps to convince the software practitioners that management is serious about process improvement. Software engineers want to do their jobs better and desire process control. The problem is that they have often been previously involved with other "quality" programs (e.g. quality circles, TQM, etc.) which have been proposed and then forgotten after a time. Process improvement must be continuously sustained on projects to be of any use. Practitioners must also be required to employ the processes put in place and management must not be allowed to drop the process in the face of crisis or setback. If this is done, software engineers will be convinced that management is serious about process improvement and make even greater effort to use the process to improve their own work. This is what we have found happening at 3M.

It should be mentioned that even though software engineers are being required to employ the standard processes adopted by the company, it is the personal desire for improvement

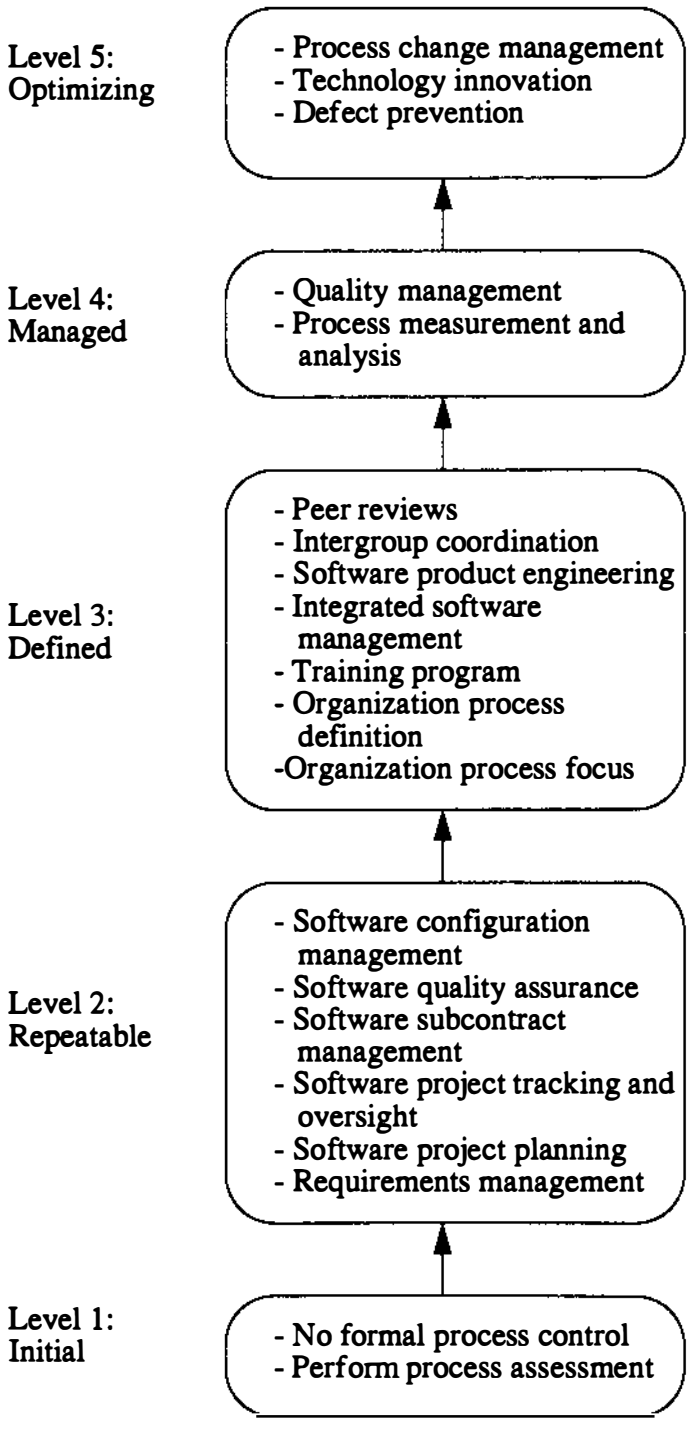


Figure 2. Maturity Level Key Process Areas

which comes from within each individual engineer that makes the process work. If the process is viewed as a hindrance instead of an aid, that desire to improve will be stifled and the process will do no good. Thus, management must make every effort to transfer ownership of the process to the engineers or personnel who use it.

Since we have not performed an assessment, we have had to deal with the training and convincing issues ourselves. Before we can discuss how we handled these issues, we must first discuss the two principles which have made our implementation of the eclectic model work. They are:

1. upper-level management wants process improvement, and
2. management has formed four groups to define the processes which must be followed at the company: a) a group to define the commercialization process for software, b) a group to define the project management process, c) a group to define the software development process, and d) a group to choose an automated project management tool.

Upper-level management desires our organization to be raised one CMM level in one year's time (we are assumed to currently be at level one; also, at that time we will be allowed to perform an assessment). Though this is ambitious improvement from an SEI perspective, it shows they are committed to process improvement (even though full resources are not now available to follow CMM explicitly). Also, with the groups formed to define process improvement, we have begun to train managers in CMM principles, since all managers must study and sign-off on the defined process procedures. When the project management tool is fully brought into the organization, both managers and practitioners will not only be trained on its use but on its use in a CMM context. All managers will eventually be required to employ the tool, and though this forced use may cause some resentment toward process control during the learning curve [Bouldin89], they will be continuing to learn CMM. The training which has been performed so far has made CMM principles become more real to users and has influenced many of these users to favor CMM usage. We have also found that success with its use in one group has almost forced other groups to employ it with success.

As management has started to use CMM processes with success, the practitioners have seen that management is serious about process improvement and have bought into the process themselves. Training of software engineers has been much more successful given this situation. This has helped to reduce the impact of process change on real work being done.

4.2 Repeatable

This section describes how the key processes from the repeatable level of CMM in Figure 2 have been implemented in the first iteration of the application of our model.

4.2.1 Requirements Management

One of the hallmarks of a chaotic software production organization is the inability to control feature requirements for a system. This lack of discipline can flow from many sources external and internal to an organization. From within a software organization, sales, marketing, service, and development can all tend to want to change feature requirements during the

development process. As a first iteration to control this process, we decided that all groups except marketing must be restricted in making changes after the requirements have been frozen. Ideally all organizations, including marketing, would be restricted from making change, but since we wish to be as market responsive as possible, for now, we are restricting only the other groups.

4.2.2 Software Project Planing

Project planning has been done at 3M, like most other companies, for years. There has been, however, no standardized project template which could be used and understood by all personnel for all projects. This has been changed by the creation of a commercialization guide which defines all the major tasks and players to take a project from conception to final release. Also, more specific guidelines for doing product development and project management have been defined in separate template guides for these processes. Now there is no question what procedures are to be followed in these activities and in what order.

Of course there is nothing new about written guidelines for software development. The difference is that management is following them and thus software engineers can follow them with the confidence that the process will give them the freedom to do their jobs correctly with fewer and fewer changes and interrupting surprises along the way.

Also, a very unique attribute of the developed process guidelines is that they have been structured to blend with the automated project management tool currently being implemented in the company.

4.2.3 Software Project Tracking and Oversight

The introduction of the automated project management software mentioned above is the most important step we have taken to promote better tracking of projects. Select managers can now simply and quickly review and update progress on projects from their own workstations. Soon all managers will have these capabilities available to them. Oversight of projects has been easier since tracking databases are updated in one place and yet accessible by all managers through a network.

Using a single tool throughout the company with a standardized project management paradigm will continue to make project management easier since all information is kept on line and managers don't have to track different projects in a different fashion.

Also with the automated system, we are gathering data which will be manipulated with automated metrics to analyze future project results (time, cost, staffing, etc.) as well as completed project problems (error causes, etc.).

4.2.4 Software Subcontract Management

We currently have no software subcontracts at our department of 3M, so we do not anticipate any process work being proposed in this area in the immediate future. This again shows an important point in our eclectic application of CMM. One should not hold the CMM model so inviolate that key processes that don't apply to a project are forced to be applied or that processes from higher levels which logically could be applied in lower levels are avoided. Our model of

application easily allows this freedom without any eventual loss of process improvement.

4.2.5 Software Quality Assurance (SQA)

At this time we have no centralized/independent SQA groups. All of the SQA activities are distributed among the various development groups. There is also no intention of changing this arrangement in the near future. Part of the reason for this is that our SQA function works very well (few errors make their way to customers), and it is hard to convince management that such a disruptive change would benefit the development process.

Of course, most processes can be improved. However, our philosophy here is that we will not address processes which are not broken until we have worked on all the ones which are. Our flexible eclectic model certainly allows this. The members of the process definition groups, however, agree with the philosophy of CMM to have an independent SQA group, and plan to work towards this goal in a future iteration of CMM application.

For now we are benefiting from having the SQA process well defined in our software development process guidelines. There has been less confusion as to the division of responsibilities between software engineering and SQA and less of a tendency to cut short testing time since the process guide is being followed.

4.2.6 Software Configuration Management (SCM)

There is no centralized CM in our department of 3M. The reasons for this are that the organization is relatively small and there are many different products being developed on many different target machines. It would be nearly impossible to accommodate all the variations in load configurations from a single CM base environment. Also, the products are produced in three entirely different application domains. No one CM group could satisfy the needs of so many disparate groups producing products for so many different target environments.

The one saving grace is that all the systems we produce are developed by relatively small groups of software engineers. As such, we have found that informal CM has proven to be adequate though potentially dangerous. As a first phase effort to introduce more structured control in the CM area, we are implementing the required use of CM tools such as UNIX's RCS.

There are also no formal configuration control boards (CCBs) in place. This is not uncommon for a small company. However in the next iteration of our model, we plan to experiment with the use of formal CCBs in a select group of test projects.

4.3 Defined

This section describes how the key processes from the defined level of CMM in Figure 2 have been implemented in the first iteration of the application of our model.

4.3.1 Organization Process Focus

This key process specifies that an organization must determine its current strengths and weaknesses and establish a group to define a standard software engineering process for the

organization [Paulk91]. We recognized some time ago that although 3M produces software of very high quality, our ability to determine when a project would be completed, like so many other companies, was not as favorable. We believe that this problem was caused mainly by our lack of written standardized procedures for developing and commercializing products.

To solve this problem, as has been mentioned, process groups were formed to define the software production process. Often this entailed simply writing down what needed to be done when, so everyone involved with the process could work with the same process sequence, expectations, and time frame.

Several pilot groups have been working with the processes as defined in the previously named documents and these processes will be amended per the results of the initial groups. Initial results indicate that of all CMM process areas being implemented at 3M, this is the one area that has shown the most positive results. We are directly attributing fewer schedule slips, higher productivity (due to less project confusion), and fewer errors shipped to standardized and documented project execution. This, more than any other key process has pleased management to the point that further process change has been approved.

As a next phase of the application of this process, we hope to be able to transform the process definition groups into a SEPG, a software engineering process group, as defined by SEI. We hope this group will be able to carry on the work of process definition, coordination, analysis of process data, and promoting the introduction of further process improvements so that we can further improve our methodologies.

4.3.2 Organization Process Definition

This process is based around the idea of standardizing the software process definition for an organization [Paulk91]. Again, we have done this with: a) our commercialization process, b) our software development process, and c) our project management process documents. As was mentioned, these processes are being used by select projects and are being phased into the entire organization. Their use is being facilitated by employing the previously discussed project management software which allows the integration of our own process control.

4.3.3 Training Program

The company has instituted a program which requires a minimum of forty hours of job-related training a year for each employee. This training covers such areas as learning to use current and future software tools and techniques, how to do problem solving, principles of CMM, and formal academic courses. There are currently no plans to modify these requirements for the next iteration of our model.

4.3.4 Integrated Software Management

This process requires that the planning and management of all software projects be based upon the organization's standard software production process [Paulk91]. As has been said, we are working towards this by gradually requiring (with poor performance rating for non-compliance) all persons to follow the standard process documents. Again this will be facilitated

by having the processes integrated into the project management tool set which we are working towards fully using.

We should mention that requiring the use of the standardized process has not gone without problems. Software production is hard enough without being forced into possibly different or new working procedures while deadlines are looming. Revolt is not uncommon if training is lacking and if a solid track record of example successes is not established before general application of the standard processes and tools.

4.3.5 Software Product Engineering

This component specifies that the software engineering process which will be followed for a project should be well defined and understood at the requirements and design stage of the project [Paulk91]. At 3M we have defined a software production process guideline which specifies that the entire project be planned even before requirements are fully defined. All software groups are moving towards the use of this guideline. Another important step that 3M has taken to relieve much of the burden placed on disparate groups to research and keep abreast of emerging technologies was the creation of what is called the Technology Forum. The Technology Forum is a group of the most advanced technical engineers from all areas of the department. They study different technologies and standardize the use of software technologies and products for software development groups in the facility. Given this, it is much easier to plan software projects since many technology issues have already been decided upon before a project even begins.

4.3.6 Intergroup Coordination

Since the majority of software products we produce are relatively small, they are usually produced by a small team of engineers in a single group. Thus, there has been little need in the past for intergroup coordination. In the future if this becomes an issue, we will revisit this process and implement it appropriately. Again, we appeal to our eclectic model to allow the lack of concentration in this area. As the company grows and larger projects are attempted, project management in this area will almost certainly be necessary.

4.3.7 Peer Reviews

Again, with the projects being small and the groups doing development consisting of relatively sparse teams, there has traditionally been no formal peer review of work being done (there are always informal group reviews performed for specifications, designs, and code). For this iteration of the model, we have specified that formal specification and design reviews be done. We would like code and test reviews to be performed as well, but at this time we feel it would be too disruptive for engineers to be required to formally perform these given the tight completion times that they are already working under. We have implemented enough disruption in other work areas, and we wanted to avoid a rebellion towards reviews. We also feel that this is not so much of a sacrifice since we wish to catch errors early (with formal specification and design reviews) in the process rather than after coding has been done.

Certainly in a subsequent iteration of the model, we will begin to institute formal

code and test reviews. Even though errors at this stage of development are expensive, letting those errors get to customers is much more expensive, and we feel that reviews are a relatively inexpensive way to avoid that.

4.4 Managed

This section describes how the key processes from the managed level of CMM in Figure 2 have been implemented in the first iteration of the application of our model.

4.4.1 Process Measurement and Analysis

At level four, an organization's standardized software development process should be stable and under statistical quality control [Paulk91]. What this means is that the development process should be so well understood and standardized and past experience with its use should be so well documented that one can very easily and accurately make estimates of future project performance by analyzing the performance of past projects. Bringing a project under statistical quality control means that one continuously gathers data concerning the performance of process related tasks and analyzes this data using metrics to determine if process goals are being met.

Before the introduction of automated project management tools, it was not easy to gather and analyze process data at our company. At this time we are preparing to institute an organization-wide program of gathering process data with the tools. We are also studying the metrics which are provided by the tool set as well as other currently accepted metrics to be employed to analyze the recorded data. In the next iteration of our eclectic model, we plan to begin the actual analysis of project data. By then we hope to have enough data gathered from trial projects to begin to draw useful conclusions.

4.4.2 Quality Management

This key process area deals with the establishment of measurable goals and priorities of product quality for each software project [Paulk91]. Also, the process is modified as appropriate to bring actual organization results in line with its goals.

At our company, stringent quality requirements and goals have been set for the organization by upper management. This is part of the reason why we have been defining software production processes, obtaining technology, and striving to achieve a new CMM level of software production ability.

In actuality, this is the process that management is most interested in controlling or achieving. They wish to be able to set software production attribute goals (error rates, completion times, numbers of engineers, etc.) and then be able to manipulate the process such that project results fit closely with these goals.

We have not yet fully achieved this ability, though we are setting the basis for its fulfillment. From the SEI's perspective, it could take upwards of a decade to achieve a organization-wide level four if starting at level one. We do not feel, however, that we can wait that long to achieve this ability to some degree. Here again, we feel that the eclectic model of CMM application will allow us to begin to experiment now with modifying the process to help achieve

project goals. In the next iteration of the model application, we plan to do this with the automated processes which we are building into our project management software.

4.5 Optimizing

This section describes how the key processes from the optimizing level of CMM in Figure 2 have been implemented in the first iteration of the application of our model.

4.5.1 Defect Prevention

In this key process, sources of product errors are identified in either the product itself or in the process which generated the product [Paulk91]. 3M has been performing these kind of activities for years as far as products are concerned. Pieces of software that are particularly faulty are redesigned, rewritten, or replaced. Analyzing the process under which faulty software is developed has been considerably more difficult to perform. This is partly due to there not being a standardized product development process and partly due to the fact that for small development organizations there is rarely time nor money to perform activities which do not directly contribute to the completion of a project, no matter how important they may seem.

With the data we are beginning to gather with our automated tools, we hope to be able to analyze the processes which we are and will be employing to determine which parts of those processes are error prone and why. This will have to wait for a future iteration of our model application, however.

4.5.2 Technology Innovation

This key process deals with the orderly and timely introduction of technology into an organization which is consistent with the standardized software production processes and is directly tied to increasing quality and productivity [Paulk91].

Above we briefly discussed the Technology Forum group that we have at 3M which has the purpose of evaluating and adopting standard and new technology for our entire department of the company. This group has proven to be very valuable as it has standardized machine platforms, operating systems, languages, DBMSs, CASE tools, and standards. Increasing productivity and quality has always been the charter of this group, so we feel that we are well on our way to performing this process. In the very least, we have already shown a reduction in project ramp-up time by having so many time consuming decisions made before a project begins.

In the future, however, we plan to be more cognizant of the interaction and impact of technology on our production processes, and vice-versa. We feel that the correct choice and introduction of technology is an important step in being able to successfully perform our processes.

4.5.3 Process Change Management

With this key process, software practices, subprocesses, and tools are continually evaluated, goals are set for their improvement, and they are modified to effect the improvement [Paulk91].

In a simplistic fashion, by the sheer fact that we have begun the work of defining and documenting a software production process, we are attempting to improve it. In a more germane fashion, however, since the software process documents were first composed and several test groups began to work with them, there has been continuous evaluation and revamping of the documents. The groups which originally produced the documents have continued to refine the processes defined within them per the results of the evaluating groups. Those activities have in turn fed into the development organization to improve their processes.

As has been said, eventually we hope to form a SEPG from a subset of the process definition personnel to continue the never ending work of evaluating and refining the process so that we will be able to progress toward a level five as an organization. With this, we feel assured that continuous improvement will be possible.

5. Conclusions

We feel that employing the eclectic model of CMM application has been a successful endeavor at 3M. The model allows us to take advantage of the benefits of CMM process improvement without having to worry about following the SEI program explicitly. It also allows us to leverage processes that we already have in place no matter what particular CMM level they contribute to.

It is difficult at this early stage of our application of the model to quantifiably determine the benefits that the process has had on our actual productivity, quality, and costs to date. However, we have begun to see a trend in schedules being more closely met, higher productivity (more projects being worked on with the same number of employees), and fewer errors getting to customers. Also, from a personnel point of view, we have received numerous positive responses to the process formalization efforts which we have made. We have as yet received no negative responses to the idea of infusing greater discipline and standardization into our process, i.e., no one likes to work in a state of confusion or unknown. Using the eclectic model to pragmatically bootstrap CMM principles into our processes has begun to eliminate surprises from our efforts. This in turn reduces stress in our engineers and provides them with more time to do productive work which actually contributes to the successful on-time completion of projects.

From an empirical point of view, we are monitoring such project attributes as number of errors produced, time to completion, customer satisfaction ratings, engineer satisfaction ratings, costs, etc. Although we expect our plan for CMM level advancement will require several years of effort (as SEI professes) before we can definitively show amelioration, the improvements mentioned above indicate that we are headed in the right direction. Also, an increase in morale brought on by introducing greater order into our software production process has been interpreted as a positive sign of improvement.

Bibliography

[Boehm88] B. W. Boehm, "A spiral model of software development and enhancement," *IEEE Computer*, May 1988.

[Bouldin89] B. M. Bouldin, *Agents of Change: Managing the Introduction of Automated Tools*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Crosby79] P. Crosby, *Quality Is Free*, McGraw-Hill, New York, 1979.

[Curtis92] B. Curtis, "Maintaining the Software Process," in *Proc. of the Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 2-8.

[Deming82] W. E. Deming, Out of Crisis, MIT Center for Advanced Study, Cambridge, Mass., 1982.

[Humphrey91] W. S. Humphrey, T. R. Snyder, and R. R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, July 1991, 11-23.

[Osterweil87] L. Osterweil, "Software Processes are Software Too," in *Proc. of the 9th International Conf. on Software Eng.*, IEEE Computer Society, Washington, D.C., 1987, pp. 2-13.

[Paulk91] M. C. Paulk, B. Curtis, M. B. Chrissis, et al, Capability Maturity Model for Software, Software Engineering Institute, Carnegie Mellon University, SEI-91-TR-24, Aug. 1991.

[Weber91] C. V. Weber, M. C. Paulk, C. J. Wise, J. V. Withey, Key Practices of the Capability Maturity Model, Software Engineering Institute, Carnegie Mellon University, SEI-91-TR-25, Aug. 1991.

A Software System Documentation Process Maturity Approach to Software Quality

Marcello Visconti
Curtis Cook

Computer Science Department
Oregon State University
Corvallis, Oregon 97331-3202

visconm@cs.orst.edu
cook@cs.orst.edu

Abstract

Our proposed solution to the problem of low quality software system documentation is to improve the documentation process. We have developed a 4-level Software System Documentation Process Maturity model based on the Software Engineering Institute (SEI) Software Process Maturity and Capability Maturity models. In this paper, we describe an assessment questionnaire and how we intend to validate our model. We give an example of the results of an assessment we conducted at a software company. We expect our model and its associated assessment questionnaire to show that the higher the maturity level, the larger proportion of design and requirement defects discovered before integration testing.

Keywords: software quality, documentation quality, documentation process, process maturity models.

About the Authors

Curtis R. Cook is professor of computer science at Oregon State University. He received his Ph.D. in computer science from the University of Iowa.

Professor Cook has over ten years of research and experience in the software field, with numerous publications and conference presentations. He has taught workshops on software complexity metrics and software quality. He is a member of the editorial board of the Software Quality Journal and has served on the program committee for several software quality conferences.

Marcello Visconti is a Ph.D. student in the computer science department at Oregon State University. He has a Masters Degree in computer science from Universidad Santa Maria in Chile. He is a professor at Universidad Santa Maria. His research interests are in software quality and software engineering.

1 Introduction

One basic goal of software engineering is to produce the best possible working software along with the best possible supporting documentation. Empirical data shows that software documentation products and processes are key components of software quality [Card87, Lien81, Romb87]. These studies show that poor quality, out of date, or missing documentation is a major cause of errors in software development and maintenance. Virtually everyone agrees that documentation is important; however, in spite of these studies, they do not fully realize that documentation is a critical contributor to software quality.

An earlier paper [Visc93a] described a maturity model to address the problem of low quality and/or missing documentation. The 4-level Software System Documentation Process Maturity model was influenced by the SEI Software Process Maturity and Capability Maturity models. Documentation refers to the system documentation generated during software development and not to end-user documentation. We have developed an assessment questionnaire that will allow us to determine the documentation process maturity level and to specify key practices and challenges to move to the next higher maturity level. We have received industry input about the adequacy and completeness of the questionnaire and have conducted a few preliminary assessments of teams at both large and small companies. This paper also describes how we intend to validate the model and its associated assessment procedure. The paper is structured as follows: section 2 provides a brief overview of our Documentation Process Maturity model and presents a summary table. Section 3 describes the assessment questionnaire, explains the scoring method associated with it and gives an example of the Documentation Assessment Report. Our validation approach is given in section 4. Section 5 analyzes what has been done and discusses future research steps.

2 The Software System Documentation Process Maturity Model

A solution to the problem of poor quality, out of date or missing documentation is to improve the documentation process. We have designed a Software System Documentation Process Maturity model that provides the basis for an assessment of the current documentation process and identifies practices and challenges to improve the process. The focus is on the documentation used in software development and maintenance and does not consider end-user documentation. Hence the term documentation refers to software system documentation. Our approach has been influenced by the SEI's Software Process and Capability Maturity models [Humph88, Paulk91]. A summary of our four-level Documentation Process Maturity model is given in table 1. The four column headings are the names of the four levels. The rows give the key features and properties of each level. Level 1 represents an ad-hoc, chaotic situation regarding documentation. Level 2 recognizes that documentation is important and there is a type of check-off list to ensure that all documentation is done. However, there is no consistent assessment of the quality of the documentation. Level 3 incorporates quality assessment of the documentation. Level 4 is attained when measurements of the quality of the documentation are fed into a process of continual improvement. A full description of the model can be found in [Visc93a].

3 Assessment Procedure

The purpose of the assessment procedure for a model is to determine where an organization stands relative to that model. For our model, the assessment maps an organization's experience and past performance to a documentation maturity level and generates a documentation process profile. The profile indicates key practices for that level, what practices the organization is doing well, what practices need improvement, and challenges to move to the next level.

Our assessment approach has been highly influenced by the SEI models. This section describes the questionnaire we have developed as part of the assessment procedure. As background we review SEI and other process assessment procedure efforts. We describe our assessment questionnaire and provide a level-by-level list of the questions, identify the relevant documentation process practices addressed in each level, and describe the scoring method used to determine documentation maturity levels. A complete and detailed description of the assessment methodology is given in [Visc93b].

	Ad-hoc	Inconsistent	Defined	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 Process Areas	Ad-hoc process Not important	Inconsistent application of standards	Documentation quality assessment Documentation usefulness assurance Process definition	Process quality assessment and measures
Key Practices	Documentation not used	Check-off list Variable content	SQA-like teams for documentation quality and usefulness Consistent use of documentation tools	Minimum process measures Data collection and analysis Extensive use of documentation tools and integration with CASE tools
Key Indicators	Documentation missing or out of date	Standards established	SQA-like practices Consistent use of documentation tools	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 Continual striving for optimization

Table 1: Documentation Process Maturity Model—Summary Table

3.1 Assessment Approach Background

The SEI developed an assessment questionnaire with the purpose of assessing software organizations' capabilities and identifying the most important areas of improvement [Humph88, Humph89]. The SEI Software Process Maturity Model Assessment Questionnaire is fully described in [Humph87], along with descriptions of the technical approach, self-assessment usage guide and guidelines for evaluation of results. The SEI questionnaire consists of 101 questions organized in the following way:

- Organization and Resource Management (17 questions): Organizational Structure (7 questions); Resources, Personnel and Training (5 questions); and Technology Management (5 questions).
- Software Engineering Process and its Management (68 questions): Documented Standards and Procedures (18 questions); Process Metrics (19 questions); Data Management and Analysis (9 questions); and Process Control (22 questions).
- Tools and Technology (16 questions).

The assessment procedure has been the target of criticism [Boll91, Humph91a]. The main criticism of the assessment procedure is the excessive role assigned to the numeric maturity level as there is little inherent value in the number itself. The assessment does not explicitly identify areas of improvement. These problems have been addressed in the new SEI Capability Maturity model [Paulk91, Weber91], which deemphasizes the numerical score and emphasizes the identification of key process areas, key practices and key indicators. It produces a key-process-area profile, rather than a single numeric maturity level. This modification of the assessment procedure allows a more explicit identification of areas of improvement, since each process area is now judged according to the level of satisfaction.

The SEI Capability Maturity Model uses the same questionnaire, but the questions were reordered to better serve the ideas of key process areas, practices and indicators introduced in the model.

Gilchrist [Gilch92] adapted the SEI Process Maturity model for individual projects and used it to assess a set of projects over an extensive period. They proposed using a spreadsheet to store and process the information, and analyzed some merits and difficulties with the approach.

Seddio [Seddio92] describes the application of review and product metrics to the software engineering process at Eastman Kodak. Reviews were held for software requirements specification and high level design, and for software programs, subsystem test plans and code coverage results. The assessment tools employed were a software requirements specifications checklist to determine documentation, completeness and testability ratings, and a code review measurement matrix. They claim that the use of this simple software assessment approach has resulted in enhanced understanding of the software process, more objective assessments of the quality of the software products, and more objective software products reviews.

Henry and Henry [Henry92] propose a methodology for assessing the software process used by an organization, integrating the principles of Total Quality Management (TQM) and the work of the SEI. The basis for integrating these 2 approaches is that the SEI assessment specifies the activities comprising the software process, and the TQM implementation phase evaluates the effectiveness of the activities. The 4 steps of their assessment methodology are: investigate the process, model the process, gather data, and analyze data.

In the following we describe our assessment approach that has been influenced by the work described above, particularly by the work of the SEI.

3.2 Questionnaire

Definition of basic terms that are extensively used:

- Software Documentation: Includes all documents generated as part of software development, i.e. software requirements specifications, design documents, code, test plans and history (test cases). Software documents refer to either hard copy or electronic form. It excludes end-user documentation, i.e. user manuals and operations manuals; and it excludes managerial documentation, i.e. project plans, schedule and staff plans, etc.
- Software CASE tools: Software designed to assist software engineers and programmers cope with the complexity of the process and the artifacts of Software Engineering.

- Documentation tools: Software designed to aid software engineers to cope with the complexity of the process and artifacts of documentation.

  Types of documentation tools:

  - Simple:
    1. Text processors: word processors, desktop publishers, editors, spelling checkers, electronic mail.
    2. Graphics: flowcharting, technical drawing.

  Advanced:

    1. Document management systems: storage, retrieval, browsing, distribution, sharing, consistency checking.
    2. Integrated documentation/CASE tools.

- Quality of documentation: Quality includes the following characteristics: adequacy, completeness, usability, consistency, currency, readability, ease of use, ease of modification, traceability.

- Usefulness of documentation: Extent to which the documentation products are used by software developers and maintainers, the documentation users.

The questionnaire we have designed consists of 56 questions, 35 of them are to be answered in the range of 1 through 5, and 21 are yes/no questions. The numbers in the range of 1-5 represent *how often* a given action is performed. The following table matches a number with its meaning.

1	never
2	seldom
3	sometimes
4	usually
5	always

The 56 questions can be decomposed as follows:

- By maturity levels:
  - Level 1 : 6 questions
  - Level 2 : 10 questions
  - Level 3 : 23 questions
  - Level 4 : 17 questions

- By subject area: (actual order of questions on questionnaire)

  Lifecycle documentation : 10 questions
  - Documentation standards : 4 questions
  - Documentation tools : 4 questions
  - Documentation quality control : 16 questions
  - Documentation usefulness determination and assurance : 4 questions
  - Documentation error analysis and improvement feedback : 13 questions

  Documentation-related training : 3 questions
  - Management attitude towards documentation : 2 questions

In the following we present the questions according to the 4 maturity levels in the Software System Documentation Process Maturity Model. The questions that are to be answered in the yes/no mode are **highlighted**.

- Level 1: Ad-hoc

  Questions at this level attempt to assess management's attitude toward documentation and determine how consistently basic system documents are created.

  1. Are software documents other than code created during development?
  2. Are software requirements specifications (including prototyping documents) generated?
  3. Are design documents (including prototyping documents) generated?
  4. Are test plan Documents generated?
  5. Are test cases used in testing recorded in a document?
  6. **Does management have a policy (not necessarily written) supporting the importance of software documentation?**

- Level 2: Inconsistent

  Questions at this level attempt to assess existence of a mechanism for ensuring that all documents are created and kept up to date. Does management truly believe documentation is a high priority?

  1. Are there check-off lists that indicate which software documents must be created?
  2. Are there standards indicating what must be included in each software document?
  3. Is there any procedure or form used to specify how and when to write each software document?
  4. Is there a formal procedure used for checking that document contents satisfy standards?
  5. Is adequate time allocated to develop software documentation during software development?
  6. Are software documents checked to see that they have been done?
  7. For each software project, is there a person responsible for collecting the documentation?
  8. For each software project, is there a person responsible for maintaining the documentation?
  9. Are simple documentation tools (text processors, graphics) used to create and maintain software documentation?
  10. **Does management view software documentation as of major importance and have written policies to this effect?**

- Level 3: Defined

  Questions at this level attempt to assess the existence of assessment mechanisms for quality and usefulness of documentation.

  1. Are all software documents generated in the development phase used? (SRS in design, design document in coding, SRS and design in maintenance, etc)
  2. **Are software documents other than code used during development?**
  3. **Are software documents other than code used during maintenance?**
  4. **When software documents are not used, is it because they are unreliable, incomplete, or out of date?**
  5. **When software documents are not used, is it because they are not easily accessible?**
  6. **Is there a mechanism for checking that a software document has been completed satisfactorily?**
  7. If so, how frequently is it used?

8. **After a change has been made to the code, is there a mechanism for checking that all related documentation is updated?**
9. If so, how frequently is it used?
10. **After a change has been made to the design, is there a mechanism for checking that all related documentation is updated?**
11. If so, how frequently is it used?
12. **After a change has been made to the requirements, is there a mechanism for checking that all related documentation is updated?**
13. If so, how frequently is it used?
14. Is affected documentation updated after each maintenance change?
15. **Is there a mechanism to monitor the quality of the documentation?**
16. If so, how frequently is it used?
17. **Is there an independent group whose function is to assess the quality of the documentation generated in a project?**
18. **Is there a mechanism to assess the usefulness of the documentation generated in a project?**
19. Is formal training available for the use of documentation standards?
20. Is formal training available for the use of documentation tools?
21. Are advanced documentation tools (integrated, documentation management) used during development and maintenance?
22. Are common sets of documentation tools used in the different development environments throughout the organization?
23. **Is there a mechanism to foster the incorporation of advances in documentation technology across the organization?**

- Level 4: Controlled

  Questions at this level attempt to assess the existence of measurement and feedback mechanisms and optimization loop.

1. Are documentation errors and trouble reports tracked to the solution?
2. Are documentation process data and error data for software projects recorded in a database?
3. Are statistics gathered on documentation errors?
4. **Is documentation error data analyzed to determine distribution and characteristics of errors?**
5. If so, how frequently is this done?
6. **Is there a mechanism to analyze documentation error root causes?**
7. If so, how frequently is it used?
8. **Is there a documentation usage profile generated for software development?**
9. If so, how frequently is this done?
10. **Is there a documentation usage profile generated for software maintenance?**
11. If so, how frequently is this done?
12. **Is there a mechanism for users of the documentation (software developers and maintainers) to provide feedback to improve documentation usefulness?**
13. If so, how frequently is it used?
14. Are measures of usefulness of documentation collected?
15. **Is there a mechanism to feedback improvements to documentation practices or standards?**
16. If so, how frequently is it used?
17. **Are advanced documentation tools integrated with software CASE tools?**

3.3 Scoring Method

We have developed a scoring method that not only yields the maturity level, but also identifies the practices of the organization at that level. There is a set of rules for the score at each level. The full details of the procedure to determine the maturity level can be found in [Visc93b]. Here we describe each level in terms of the key practices that define it.

- Level 1: Since this is the first level in our model, no effort is needed to attain it. So, the main interest at this level is to determine whether the organization shows a strong level 1 performance.

  Strong: The key practices that define this level are: creation of software development documents and documentation generally recognized as important.

- Level 2: At this level, the organization could be in either one of three situations: weak, solid, or strong performance. The performance is described incrementally: an organization needs to satisfy the requirements for a weak, then a solid, and finally a strong performance.

  - Weak: The key practices that define this level are: use of a check-off list of required documentation and adherence to documentation standards.

    Solid: The key practices that define this level are: adherence to documentation standards and use of simple documentation tools.

  - Strong: The key practices that define this level are: written statement about importance of documentation and adequate time and resources for documentation.

- Level 3: This level is defined in terms of a solid or a strong performance.

  - Solid: The key practices that define this level are: use of software documentation generated, mechanisms to update documentation, mechanisms to monitor quality of documentation and methods to assess usefulness of documentation.

  - Strong: The key practices that define this level are: use of common sets of documentation tools, use of advanced documentation tools and documentation-related technology and training.

- Level 4: This level is defined only in terms of a solid performance, given its optimizing nature.

  Solid: The key practices that define this level are: measures of documentation process quality, analysis of documentation usage and usefulness, process improvement feedback loop and integrate CASE and documentation tools.

The responses to the assessment questionnaire are used to identify areas that need improvement at that level before moving to the next higher maturity level. The relative rating within a level is based on the responses to questions concerning key practices at that level. Partial responses indicate areas of improvement. Low responses indicate challenges.

3.4 Documentation Assessment Report

Based on the responses to the questionnaire we generate a Documentation Assessment Report that contains an executive summary with the maturity level number, a documentation process maturity profile, and an action plan. The profile indicates satisfactory practices, practices needing improvement, missing practices, and challenges to move to the next level. The action plan describes specific actions to improve existing practices and to address missing practices to enable the organization to move to the next higher maturity level. We present as an example an actual report we generated after conducting an assessment for a software project at a software company identified as project Y and company X. A complete description and a sample form of the report can be found in [Visc93b].

Example Documentation Assessment Report

Contents of the Report

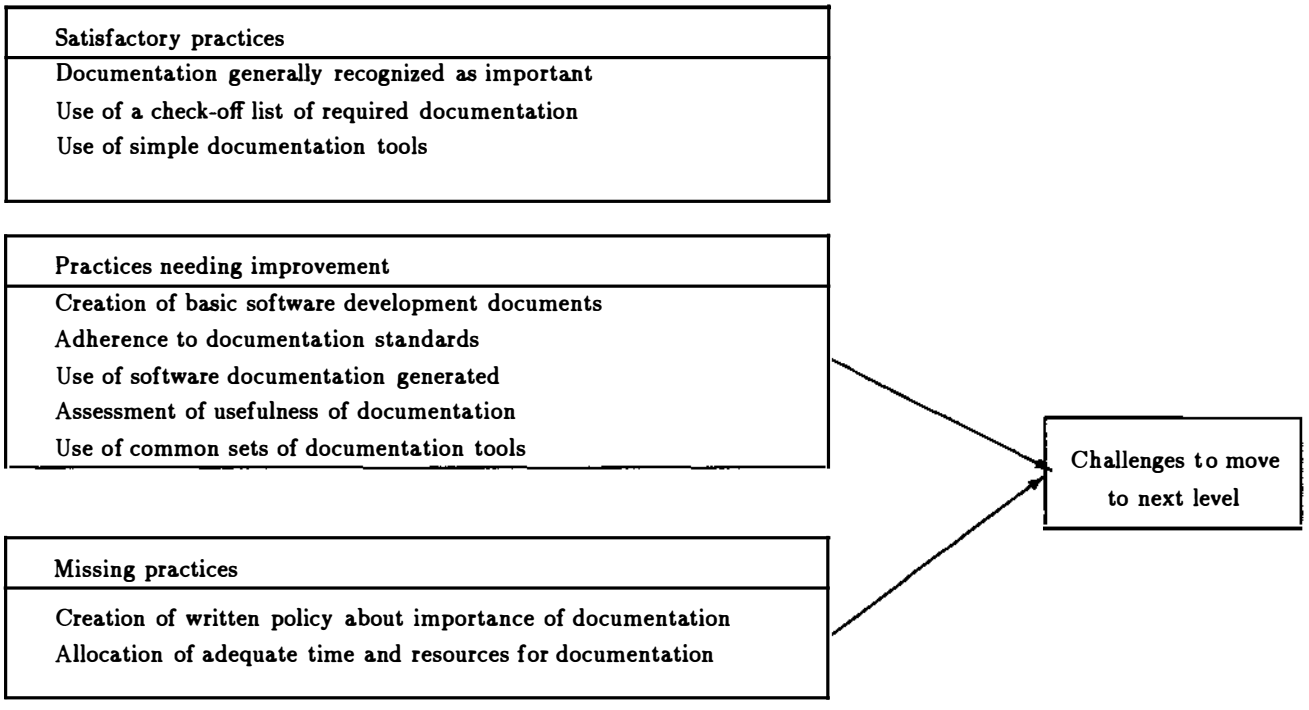
This report presents an executive summary of the main findings of the assessment conducted for project *Y* at company *X* on *month day, year*. It includes the process maturity level, a documentation process maturity profile table, and an action plan of needed improvements and challenges.

Executive Summary

Maturity Level

The maturity level for this project is: **Level 1**

Documentation Process Profile



Level	Key Practices	Degree of Satisfaction
1	Creation of basic software development documents	Partial
	Documentation generally recognized as important	Full
2	Written statement about importance of documentation	None
	Adequate time and resources for documentation	None
	Adherence to documentation standards	Partial
	Use of a check-off list of required documentation	Full
	Use of simple documentation tools	Full
3	Use of software documentation generated	Partial
	Mechanisms to update documentation	None
	Mechanisms to monitor quality of documentation	None
	Methods to assess usefulness of documentation	Partial
	Use of common sets of documentation tools	Partial
	Use of advanced documentation tools	None
	Documentation-related technology and training	None
4	Measures of documentation process quality	None
	Analysis of documentation usage and usefulness	None
	Process improvement feedback loop	None
	Integrate CASE and documentation tools	None

Table 2: Maturity Profile of Documentation Process Practices

Example Action Plan

The assessment has defined the project to be at a *Level 1* maturity level, which means that documentation is not given a high priority.
The followings lists describe the actions needed to solidify the current practices and the actions needed to move to the next maturity level.

Satisfactory current practices

- Documentation is generally recognized as important.
- Use of a check-off list of required documentation.
- Use of simple documentation tools

Current documentation practices needing improvement

- All basic software documents must be created for all phases of software development. This includes software requirements specification, design documents, test plans, and test implementations.
- Documentation standards must be followed in the creation of all software documents.
- Software documentation generated must be used in all subsequent phases of the software process.
- Interview software developers and maintainers to assess usefulness of all documents.
- Use of a common set of tools by the software developers when working in a particular software development environment.

Documentation practices that need to be addressed

- Create written policy on importance of documentation and make that policy an important part of each software development project.
- Allocate sufficient time and resources to create software development documentation that meets the standards for each phase of the development.

Example Error Data and Documentation Maturity Analysis

We were provided with error data for project Y. Our conclusions regarding the documentation process were very consistent with the distribution of defects. The error data for this project show the need for improved early error detection, especially during design: while 83% of the errors are introduced before testing, 64% of them are found during testing or after. 6% of all errors are introduced in design, but only 1% of them are found before implementation. Even though we were clearly told that the errors were not always recorded in the database, the table below is very indicative of the general situation. We strongly believe that improving the documentation process as we have outlined above should significantly increase the proportion of errors that are caught in the same phase in which they are introduced.

The following table shows the cumulative percentages of errors introduced and found by the end of every phase. The data is presented for critical and serious errors and for all errors. The data shows that by the end of design, only 1 out of 72 errors has been found, and that error was of *low severity*.

At the end of	Critical and Serious errors		All errors	
	% introduced	% found	% introduced	% found
Design	5	0	6	1
Implementation	83	39	83	36
Test	99	94	98	91
Post-release	100	100	100	100

Table 3: Cumulative Error Percentages

4 Validation of Our Model

The ultimate goal of the validation of our model is to show that organizations at a higher level produce higher quality software. For our validation we decided to concentrate on software errors because errors are a commonly accepted measure of quality and organizations are most likely to collect error data. The first part of this section shows that most software errors are not coding errors, but are documentation errors. Hence the quality of the documentation does impact the number of errors. Error removal is expensive because most errors are introduced early (e.g. requirements and design) in software development, but are not discovered and removed until much later. The second part of this section describes how we intend to collect error data to validate our model. Our goal is to show that the organizations at a high maturity level catch a larger proportion of the errors earlier in the development phase than organizations at a lower maturity level. The third part of this section briefly analyzes the SEI's approach to validation.

4.1 Errors in Software Development

Many empirical studies of errors discovered during integration testing and maintenance have shown the following:

- The majority of errors were introduced before any code was written.
- Most errors are removed long after they were introduced, resulting in very expensive testing and maintenance processes.

Studies have consistently shown that the majority of errors were introduced before coding, but discovered and removed after coding. Boehm [Boehm81] estimates that 42% of the errors discovered during testing were introduced during the program design phase. Alberts [Alb76] found that 46-64% of all errors were design errors and Grady and Caswell [Grady87] found that 50% of errors are design errors. In Basili and Perricone [Basi84] 48% of errors were attributed to misinterpreted functional requirements or specifications. Endres [Endres75] analyzed software errors and their causes, and concluded that 46% of them relate to poor communications and problem understanding. Ramamoorthy [Ramam88] states that the majority of requirements and design errors are caused by ambiguity, incompleteness, or faulty assumptions in the specifications. Grady and Caswell [Grady87] claim that documentation is responsible for 50% of design errors. Seddio [Seddio92] describes an application of review and product metrics to the software process at Eastman Kodak. He found that 63% of errors in software specification documentation are caused by incomplete specifications and that 24% violate documentation standards. Boehm [Boehm81] believes that typically there are twice as many

design errors as coding errors. All of these studies clearly indicate that most errors are design and requirements errors and in most instances these errors are discovered and corrected during testing and maintenance. Therefore, errors are introduced before coding, but not found until after the coding.

It is much less expensive to repair an error early in the software life cycle than later in the life cycle. Boehm estimates that it may cost 50 times more to discover and repair a design error during testing than during design and it may cost 100 times more if the error is repaired during maintenance. Alberts [Alb76] estimated that up to 70% of design errors are not found during development and the cost to remove these errors is almost half (47%) of the development costs. His analysis of the costs associated with errors revealed that 80% of the cost can be related to design errors.

What can be done to reduce the number of errors in the early stages of software development, most of which are system documentation errors - inaccurate, incomplete, or missing information? The most effective techniques would seem to be ones aimed at improving the quality of system documentation and the documentation process. This has been confirmed by several studies. In one experiment Fagan [Fagan76] found that 82% of the total number of errors discovered during development were found during formal design and code inspections. A study at GTE by Howden [How78] compared the efficiency of design review and testing. Design reviews uncovered 45% of the errors taking 17% of the development time, whereas testing took up to 75% of the time to uncover just 10% of the errors.

4.2 Description of Our Validation Approach

We believe that a higher level documentation process will produce a higher quality product. Since most errors are introduced before coding and high quality documentation should aid in the discovery of these errors, a higher level documentation process should lead to the discovery of more of these errors than a lower level documentation process. To validate our model we intend to show that for a software project developed with a higher documentation process maturity level, the fraction of the total errors discovered during testing that are requirements and design errors will be less than for projects developed with a lower documentation process maturity level. The fraction of total errors that are requirements and design errors was chosen rather than the total number of errors as our measure because the total number of errors depends on the type of application, the size of the application, programmer training and experience, and a variety of other factors. We believe the fraction measure best reflects the influence of the quality of documentation on errors.

To gather our empirical data we will select project teams and specific projects in an organization. Each team will complete the assessment questionnaire for a particular project and provide the error data for that project. The error data will include: number of errors introduced in each phase (requirements, design, coding, testing, maintenance), severity of each error (serious, moderate, minor), number of errors detected during each phase, and number of errors corrected during each phase. This is error data that many organizations already collect. From the assessment questionnaire the team's documentation maturity level can be determined. The error data will be correlated with maturity levels and errors to verify our hypothesis. It is expected that our data show that the higher the project team documentation process level, the smaller the fraction of requirements and design to total errors. This will enable us to estimate the cost/benefit of the higher documentation maturity levels. Clearly it is far less expensive to discover and correct errors in the phase in which they are introduced than in a later phase. We do not expect to be able to make any statements about the total number of errors being less for teams at higher maturity levels since the number of errors depends on the size of the project, application area, and so forth. However, we expect to see a different error distribution by phases for teams with higher maturity levels than for teams with lower maturity levels.

4.3 SEI's approach to validation

As part of its effort to validate its model and assessment procedure, the SEI conducted 2 process assessments for Hughes Aircraft's Software Engineering Division (SED) : one in 1987 and the other one in 1990 [Humph91]. According to the SEI, Hughes' SED moved from level 2 to level 3 in its 5-level maturity model. The 1987 assessment identified the strengths and weaknesses of the SED, and proposed some actions for process improvements. The 1990 assessment found a strong level 3 organization. SEI claims that the process improvement was tremendously cost-effective. The assessment itself cost Hughes $45,000 and the

two-year improvement plan cost about $400,000. The revenues (represented by better working conditions, employee morale, scheduling, and costs) were estimated at $2 million annual savings, by comparing the *Cost Performance Index* variation in the period.

By 1990, 14 assessments and 18 self-assessments had been conducted by the SEI.

A detailed description of the assessment procedure can be found in [Humph87, Humph91], although the SEI does not provide details about how they arrived at the costs and estimated savings figures they claim. We think that the SEI has not actually *validated* its maturity model and associated assessment procedure, but rather reported on a successful experience. They have not provided a quantitative estimation of the savings and costs of moving from one level to the next for all 5 levels. This shows how difficult it is to validate maturity models like the one we have proposed.

5 Final Remarks

This paper described an assessment procedure to complement the Software System Documentation Process Maturity model that we created to address the problem oflow quality and/or missing documentation.

The assessment questionnaire yields the documentation process maturity level, key practices, and key challenges to move to the next higher maturity level. We have conducted a few preliminary assessments. Over the next few months we intend to conduct additional assessments at industrial sites. At each site we plan on having several teams complete the assessment questionnaire for recent software projects and provide us with the defect data for the projects. To validate our model, we plan to compare the maturity levels of the groups with the number of design and requirements defects and when these defects were found and corrected. We expect our model to show that the higher the maturity level the larger proportion of design and requirement defects discovered before integration testing.

SEI models do not thoroughly address documentation issues, ours focuses on documentation. Our assessment procedure is far less resource demanding than the SEI procedure, it takes only 30 minutes to complete the questionnaire. It would be unreasonable to expect a short questionnaire to identify in detail problems in the documentation process, however it does identify process problem areas that should be investigated and studied in detail.

Acknowledgments

We thank Lisa Friendly, Eric Schnellmann, and Steve Schellans for sharing their views and comments on an earlier version of the assessment questionnaire.

References

[Alb76] Alberts, David S. The economics of software quality assurance. Conference Proceedings, NCC, New York, 1976.

[Basi84] Basili, Victor R. and Perricone, Barry T. Software errors and complexity: an empirical investigation. Communications of the ACM, Vol. 27, No. 1, 1984.

[Boehm81] Boehm, Barry W. Software engineering economics, Prentice-Hall, 1981.

[Boll91] Bollinger, Terry B. and McGowan, Clement. A critical look at software capability evaluations. IEEE Software, July 1991.

[Card87] Card, David N., Mc Garry, Frank E. and Page, Gerald T. Evaluating software engineering technologies. IEEE Transactions on software engineering, Vol. SE-13, No. 7, 1987.

[Endres75] Endres, Albert. An analysis of errors and their causes in system programs. IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, 1975.

[Fagan76] Fagan, M.E. Design and code inspections to reduce errors in program development. IBM Systems Journal, Vol. 15, No. 3, 1976.

[Gilch92] Gilchrist, J.M. Project evaluation using the SEI method. Software Quality Journal 1, 37-44, 1992.

[Grady87] Grady, Robert B. and Caswell, Deborah L. Software metrics: establishing a company-wide program. Prentice-Hall, 1987.

[Henry92] Henry, Joel and Henry, Sallie. An integrated approach to software process assessment. Proceedings of the Pacific Northwest Software Quality Conference, 1992.

[How78] Howden, William E. Empirical studies of software validation. Tutorial: Software Testing & validation Techniques, IEEE Computer Society, 1978.

[Humph87] Humphrey, Watt S. and Sweet, W. L. A method for assessing the software engineering capability of contractors. CMU/SEI-87-TR-23, ESD-TR-87-186, September 1987.

[Humph88] Humphrey, Watt S. Characterizing the software process: a maturity framework. IEEE Software, March 1988.

[Humph89] Humphrey, Watt S. Managing the software process. Addison-Wesley, Reading, Mass, 1989.

[Humph91] Humphrey, Watt S., Snyder, Terry R. and Willis, Ronald R. Software process improvement at Hughes Aircraft. IEEE Software, July 1991.

[Humph91a] Humphrey, Watt S. and Curtis, Bill. Comments on 'A critical look'. IEEE Software, July 1991.

[Lien81] Lientz, Bennet P. and Swanson, E. Burton. Problems in application software maintenance. Communications of the ACM, Vol. 24, No. 11, 1981.

[Paulk91] Paulk, M. C., Curtis, B., Chrissis, M. B., et al. Capability Maturity model for software. CMU/SEI-91-TR-24, August 1991.

[Ramam88] Ramamoorthy, C.V. Our job is to reduce the errors. From Myers, Ware. Can software for the strategic defense initiative ever be error free? IEEE Computer, Vol. 21, No. 11, 1988.

[Romb87] Rombach, H. Dieter and Basili, Victor R. Quantitative assessment of maintenance: an industrial case study. Proceedings Conference on Software Maintenance, IEEE, 1987.

[Seddio92] Seddio, Carl. Applying review and product metrics to the software engineering process: a case study. Software Quality Journal 1, pp. 133-145, 1992.

[Visc93a] Visconti, Marcello A., Cook, Curtis R. Software System Documentation Process Maturity Model. Proceedings 21st ACM Computer Science Conference CSC '93, Indianapolis, IN, February 1993, pp. 352-357.

[Visc93b] Visconti, Marcello A., Cook, Curtis R. Software system documentation process maturity model assessment methodology. Technical Report 93-60-13, Computer Science Department, Oregon State University, May 1993.

[Weber91] Weber, C.V., Paulk, M.C., Wise, C.J., Withey, J.V. Key practices of the Capability Maturity model. CMU/SEI-91-TR-25, August 1991.

The FDA and Software for Medical Devices

Dennis Rubenacker
Senior Partner

Noblitt & Rueland
5405 Alton Parkway, Suite A530
Irvine, CA 92714
(714) 857-2125

Abstract

Medical devices must have FDA approval in order to be marketed in the U.S. Software and the software development process have gained increased scrutiny in the eyes of the FDA with special emphasis on medical device software and software that assists in manufacturing medical devices. Device recalls, seizures, injunctions, civil penalties, and other FDA regulatory actions are increasing for medical device manufacturers.

Device definition, embedded device software, third party software, "off the shelf" software, software development processes, verification, validation, and other important issues will be discussed as they pertain to FDA requirements/guidelines and *new* FDA regulations for design control due next year.

Special focus will be directed toward strategies for companies that plan to provide software to the medical device and healthcare industry.

Biography

Dennis Rubenacker is a founder of the professional management and technical consulting firm and training seminar company, Noblitt & Rueland. He has extensive experience in software engineering, software quality assurance, software verification, software validation, and management. Mr. Rubenacker specializes in medical device instrumentation software and medical manufacturing process software including the development of monitoring, diagnostic, and therapeutic critical-care devices. His experience includes blood analyzers, implantable & external defibrillators, glucose monitors, pacemakers, cardiac output computers, ejection fraction computers, oxygen saturation computers, retroperfusion pumps, cardiac imagers, EKG monitors, infusion pumps, catheter-sensor interfaces, and home-healthcare monitoring devices. He has been involved in training or consulting with over 2000 representatives from over 400 international medical device manufacturers on issues concerning software development, the FDA, and ISO. Mr. Rubenacker received his B.S. in Electrical Engineering with highest honors from the University of Illinois and is a member of the IEEE, Regulatory Affairs Professional Society (RAPS), ASQC, and a founding association member of the Medical Device Manufacturers Association (MDMA).

ISO 9000-3 Self-Auditing Using the SDI Method

Vladan Jovanovic
Dan Shoemaker

University of Detroit Mercy
Computer and Information Systems
4001 W.McNichols, Detroit, MI 498221
tel 313 993 1237

Abstract

This paper presents the SDI (Sense-Diagnose-Improve) method for the internal audit of ISO 9000-3 compliance. The questionnaire is designed to clarify ISO 9000 series requirements as they apply to software production, specifically for business oriented software development and maintenance. The structure of the questionnaire is influenced by the documentation required as proof of compliance, as well as the structure of requirements as expressed in the standards themselves (ISO 9001, ISO 9000-3 and ISO 9004-2). This is necessary for registration.

Level one is a prerequisite assessment of upper management commitment. During this phase the importance and consequences of a quality system is characterized for upper management and their participation is obtained. In Level two a detailed diagnostic questionnaire is administered aimed at determining areas where quality must be improved in order to achieve compliance with ISO 9001 and ISO 9000-3. Conceptually level two is determined by the standards themselves. Level three is a vehicle to govern preparation of software process improvement recommendations. It establishes orderly baselines in compliance with the ISO 9000-3 and accepted practices of the software industry.

ISO 9000-3 Self-Auditing Using the SDI Method

Background: ISO 9000, What Is It?

Many nations are making ISO certification a prerequisite for doing business. Their intention is to exclude suppliers who aren't serious about quality. As a result, organizations must have explicit ISO certification in order to prove the ***capabilities*** that heretofore were more a function of good salesmanship. Quite understandably, the push is now on to find fool-proof ways to document a commitment to quality.

The general purpose of ISO 9000 is to improve quality by assuring a contractual basis for defining it. The ISO 9000 series consists of five documents, two guidelines (ISO 9000 Quality Management and Quality Assurance Standards and Guidelines for Selection and Use and ISO 9004 Quality Management and Quality System Elements) and three models of quality assurance systems (9001, 9002, and 9003). Software and information systems developers are instructed to use ISO 9000-3 "Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software", published in 1991. Strictly speaking, the software developer's process must conform to requirements specified in ISO 9001. Certification of conformance is actually a registration (accreditation) of an audited and accepted quality management system. ISO 9001 and the accompanying, interpretation in, 9000-3 guidelines, represent an international standard but compared to government standards like the mandatory DoD 2167A and 2168 and the voluntary industry Standards like IEEE's software engineering series, ISO 9000-3 is more like a complete quality system framework rather than an explicit directive.

Table one presents the comparative coverage of most software quality assurance standards (Shulmayer, 1992) with the addition of ISO 9001 and ISO 9000-3. The intention of ISO 9000 was to bring the various competing standards under a single standard. It actually originated in 1959, with DoD.'s Quality Management Program (MIL-Q-9858). This standard itemized a set of quality system requirements which was adopted with slight changes by NATO (AQAP-1) and used by both through the 1970s. In 1979, the British Standards Institute (BSI) developed the first quality management standard for industry (BS 5750) from the AQAP-1 standard. The general intent of BS 5750 was to define the requirements for a quality system that would fit most businesses. None of this was specifically software related. In

1987, ISO began publishing the ISO 9000 series (developed directly from BS 5750). The U.S. adopted ISO 9000 through the Q-90 standards published by the American National Standards Institute (ANSI) and the American Society for Quality Control (ASQC).

Under normal circumstances a company can use preparation for an ISO 900x audit to assess, consolidate and improve their quality assurance system as a means of quality process management. This paper presents a model which addresses all elements of ISO 9001 requirements as clarified by ISO 9000-3. The Model can be used in two ways; as a pre-assessment instrument to prepare for certification, or for the development of software process improvement recommendations.

Working Hypotheses

We have developed explicit and detailed guidelines that will allow software managers to evaluate the quality of their software process along with the state of their compliance with ISO 9001, ISO 9000-3 and ISO 9004-2. We believe that such an assessment will help the software organization improve itself by identifying critical problems as an essential first step in determining their solution. The result is a better educated, organized and more responsive organization. Our work is based on three general hypotheses:

- **Software Process Quality can be defined, understood and communicated (ISO 9001, ISO 9000-3 and SDI level one)**
- **Software Quality can be measured and these data can be used to to direct process improvement activity (SDI level two questionnaire)**
- **Software Quality is attainable because it can be improved and verified (SDI level three questionnaire)**

The Sense-Diagnose-Improve (SDI) Model

Background: Other Approaches to Process Quality Assessment

The current models for software process quality assessment pose a number of practical issues. These include:

- **how complete is the coverage**
- **what is the theoretical basis of the model**
- **what is the basis for the selection of the detailed questions**
- **how will any of this be verified**

Through the following brief discussion we hope to demonstrate where the SDI model fits with other industry recognized approaches to software process quality assessment.

	SUBJECT MATTER	MIL-S-52779	FAA-STD-018	DI-R-2174	MIL-STD-1679	MIL-S-52779A	ANSI/IEEE STD 730	UE-759/ESD	AQAP-13	ANSI/IEEE STD 730-1981	MIL-STD-SQAM	RADC-TR-126	DI-R-1105	DOD-STD-1679A	DI-R-2174A	EEE STD 730-1984	DOD-STD-2167	DOD-STD-2168	DI-MCCR-80010	DOD-STD-2167A (DRAFT)	DOD-STD-2168A	DI-MCCR-80030A	DI-QCI-80572	MIL-HDBK-286	DOD-HDBK-287A (DRAFT)	MIL-HDBK-287A (DRAFT)	IEEE 1012-1987	ISO 9000-3
1	Organizational Structure	X	X	X	X	X	X		X	X			X	X	X	X		X	X	X	X	X	X	X	X	X	X
2	Personnel Required				X								X	X				X	X	X	X	X	X	X	X	X	X	X
3	Resources Required				X			X			X		X	X				X	X	X	X	X	X	X	X	X	X	X
4	Schedules		X		X						X	X		X				X	X	X	X			X		X	X	X
5	Evaluation Procedures	X	X	X	X	X	X		X	X	X		X	X	X	X	X	X	X	X	X	X		X		X	X	X
6	Evaluation of Development Processes	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X		X		X	X	X	X
7	Configuration Management	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X		X		X	X	X	X
8	Software Development Library	X	X		X	X	X	X	X	X				X		X	X	X	X	X	X		X		X	X	X	X
9	Documentation Reviews	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	X	X	X
10	Evaluation of Media Distribution	X	X		X	X	X			X	X	X		X		X	X	X	X				X		X		X	X
11	Storage and Handling	X	X	X		X	X			X	X	X			X	X	X	X	X		X		X		X	X	X	X
12	Evaluation of Non-Deliverables					X					X						X	X	X				X		X		X	X
13	Evaluation of Risk Management				X									X			X	X	X	X	X					X	X	X
14	Subcontractor Controls	X	X	X	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X	X		X	X	X	X
15	Reusable Software Controls								X								X	X	X	X	X			X		X	X	X
16	Records		X		X	X	X			X	X			X		X	X	X	X	X		X	X	X	X		X	X
17	Corrective Action	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X		X		X	X	X	X
18	Quality Evaluation Reports		X	X	X						X	X	X	X	X		X	X	X	X	X		X		X	X	X	X
19	Certification			X	X			X			X	X		X			X	X	X				X		X			X
20	Software IV & V Contractor Interface				X									X		X		X	X	X	X					X	X	X
21	Government Facility Review	X	X		X	X								X				X	X				X		X			
22	Quality Cost Data																X	X	X									X
23	Activities Evaluation	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	X	X	X
24	Products Evaluation	X	X	X	X	X	X		X	X	X	X		X	X	X	X	X	X	X	X		X		X	X	X	X
25	Software Test and Evaluation	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	X	X	X
26	Acceptance Inspection				X				X		X	X		X		X	X	X	X	X	X		X		X	X	X	X
27	Installation and Checkout																X	X	X		X					X	X	X
28	Code Reviews			X	X	X	X			X	X	X	X	X	X	X	X	X	X	X	X					X	X	X
29	Deviations and Waivers										X								X									
30	Work Tasking, Authorization, Instructions	X	X		X	X	X		X	X				X	X	X												X
31	Accommodations and Assistance								X																			
32	Evaluation Criteria																		X	X	X		X		X	X	X	X
33	Transition to Software Support																			X	X					X	X	X
34	Safety Analysis																			X	X					X		
35	Software Development File																			X	X	X	X		X	X	X	X
36	Non-developmental Software																			X	X		X		X	X	X	X

Table One. Software Quality Assurance Standards Comparison

Software Capability Assessment

The Software Engineering Institute (SEI) Process Assessment establishes a baseline to determine the software process maturity level of an organization (Humprey, 1987 and 1988), with extensions of Capability Maturity Model Version 1.1 reported in (Paulk et al. 1993). It is not an audit but a review, typically done by a team of software professionals. The basic approach is to conduct a structured series of interviews using a questionnaire. Since its inception, this method has been extensively discussed and improved (Baumert 1991, Bollinger 1991, Humphry,1991). The Bootstrap method has been implemented in Europe to expand on the SEI model (Bisego et al., 1993). This approach refines and expands the SEI model by using ISO 9000-3 requirements to produce sub-levels and to identify areas out of compliance. A software capability assessment (such as SEI provides) is a reasonable prerequisite for ISO 9001 certification. Only an organization having a **repeatable** process, or any higher model of maturity (see table two), can attempt ISO 9001 registration. Even then this should come as a part of a deliberate effort to make the software process **definable**, or **manageable**.

Comprehensive Software Metrics and Improvement Programs.

Questions developed by Jones (Jones, 1986) were instrumental in establishing the industry's first widely recognized metrics program at Hewlett-Packard (Grady, 1987, and 1992). In the much larger market of software production to support business operations, assessments oriented toward measurement of information system value were developed first by IBM in 1987 (as reported by Carlson et al., 1989). This synthesized earlier work done under the auspices of IBM in the area of Information Economics (reported by Parker in 1988), Function Point Analysis (Albreht, 1983) and others. A general measurement framework has recently been proposed comparing IBM's efforts (Carlson, 1992) with ISO 9000 requirements and characteristic criteria from the Malcolm Baldrige quality award (Steeples, 1993). The work of Ruben (Ruben, 1991) in developing benchmarks and measurement systems for the software business is also a significant contribution.

Specific process oriented approaches

Other work in the area of questionnaire development for the purpose of software process quality assessment focuses on specific processes or market points of a view. Numerous approaches have contributed such as: Hetzel's Diagnostic Checklists and his work on testing practices (Hetzel, 1987), and work done by Perry (Perry 1991) on quality assurance for information systems. There has also been work done on software reviews and audits (Hollocker 1990). All of these are explicitly defined in a software quality

improvement context. Their work, and many other works with this specific focus can be seen as complementary to our third level process questionnaire. They can be used either as alternative methods when specific area improvement is desired. Or (because our efforts are still evolving), they can be valuable supplements to be adopted where specific sub-processes improvement is desired.

Our pre-assessment questionnaire is designed to help companies (which we will call Units) discover their status regarding ISO 9000-3 and lets them focus on making substantive improvements. The importance of the pre-assessments phase is recognized in ISO 9001 which requires systematic internal audits before certification can be awarded.

The ISO 9001 "Model for Quality Assurance in Design/Development, Production, Installation and Servicing" combined with the ISO 9000-3 "Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software", is applicable in contractual situations when suppliers must demonstrate their capability to design, develop, produce, install, and service software. It is our assertion that the SDI model can be specifically tailored to fit the business application development segment of this market and even be extended to areas where a contract doesn't exist (e.g., internal development). This includes ordinary business organizations producing or acquiring software to support operations that will eventually be certified as part of an ISO 9001 audit. It should be noted that we have specifically excluded military software development and acquisitions (which is the province of SEI).

The following general principles guided the process of mapping and sequencing our questionnaire for the target market defined above. They are specifically intended to deal with problems associated with assessment models.

1. The questionnaire was stratified into a hierarchy of 3 levels to address the different responsibilities of differing types of professionals: e.g., high level managers, quality professionals, and software professionals

2- Information gathering is treated systematically. The questionnaire is designed to support process improvement. It explicitly serves the contextual purposes of motivation, diagnosis and benchmarking.

3- Pre-assessment results are verified. A single verification model is provided which can be universally applied to assess compliance with ISO 9001 and ISO 9000-3 requirements.

4- There is a theoretical foundation for every conceptual element in the assessment. Every process can be realistically described in terms of the

four phases that comprise the life cycle: conception, - birth, - growth- death (a new beginning of a cycle).

Which gives us five ending points from which we can model the five distinct stages in the process of evolution. These are identified by Roman numerals in **Table 2** which illustrate three popular five stage frameworks. This five level schema can be traced back to Cosby's (Cosby 1978) Quality Maturity Model.

Table 2: Technology Sophistication Levels I through V

	People's Competency	Process Model Maturity (Humprey)	Software Quality (Shulmeyer)
I	**Terminology**	**Initial**	**SQ not recognized**
II	**Relationships**	**Repeatable**	**SQA in crisis**
III	**Analysis**	**Defined**	**SQA Planned**
IV	**Design**	**Managed**	**SQA Management**
V	**Evaluation**	**Optimized**	**Quality Software 100%**

There are three generally accepted ways of organizing an assessment of software. It can be assessed as a **Product,** a **Project** or a **Process.** ISO 9000 explicitly uses process quality as the driver. This creates a set of requirements aimed at assuring project and product integrity. The assumption behind this is that only a quality process can produce a quality product.

What comprises the SDI method

The three components that we believe are absolutely critical to the process are incorporated in this model. These are: Motivation, Diagnosis and Improvement. Each of these parts is implemented by its own assessment instrument. Verification is defined generically, and a procedure is provided to audit compliance. This general verification framework encompasses all elements of the quality requirements which are the focus of ISO 9001, guaranteeing, as a minimum, the satisfaction of requirements stated by ISO 9000-3.

SDI Level I: The Sensing Level

The questionnaire is divided into three levels which are to be executed in a sequence. The first part is a list of five motivation oriented (sensing) questions which indicate the extent of management control and participation. These are to be answered as expeditiously as possible in an

essay format by the top-level executives of the Unit. The rationale for this sort of free-form response is that executives are allowed to indicate their awareness of the potential benefits of a quality system as well as their willingness to actively support it. There is no correct answer to these questions. However, it is possible to get a general sense of the likelihood that a quality system can be implemented and maintained. What follows is a summary of the topics addressed by this first stage.

Level I Questions are:

Sensing question 1: What represents quality for your customers? Do you have a systematic mechanism to quantify what value a customer obtained from the activities of your Unit? (no reference is available)

Sensing question 2: How do you measure the value of software assets? How do you cost-out quality (Carlson et al., 1992)? Note for example that the estimated cost of quality (Marchinak 1990) adds 5% to the overall cost of software development.

Sensing question 3: Do you have a formal model for the software process, represented in a quality assurance context (Humprey 1988, Ould 1990)?

Sensing question 4: Do you have a current Index of Quality System Documents (Daily 1992)?

Sensing question 5: What level of executive participation in quality improvement do you expect (no reference is available)?

SDI Level II: The Diagnostic Level

Level II presents a list of questions which explicate (diagnose) the required process areas (grouped into three main categories). Each of these areas has five True-False questions. These areas of concern and their groups are derived directly from the ISO 9000-3. The underlying assumption is that direct tracking of ISO 9000-3 compliance can be accomplished by following a consistent verification process (logically, we must assume that the standard itself is verifiable, otherwise no compliance registration would be feasible). To represent this software process structure graphically, we decided to adopt the Rolling Hexagonal Verification Model developed by Ould (Ould, 1990). What follows is a detailed checklist (the Level II Diagnostic Checklist) of elements to be used, when applicable, to characterize the process under assessment (defined per ISO requirement).

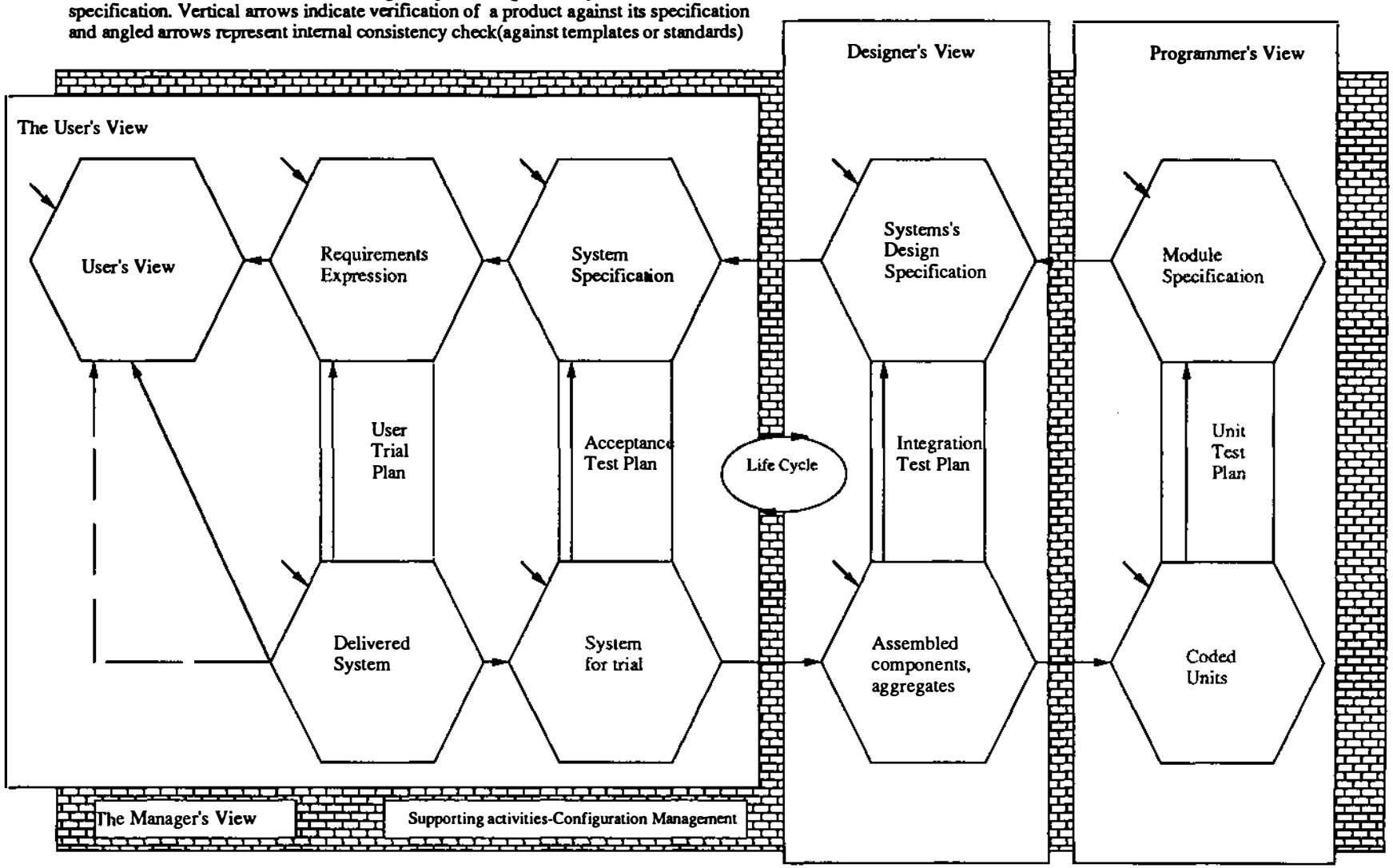


Figure one. Verification Model of a Software Process

This list represents our recommendation for the set of elements which should be used to describe the implementation, and improvement of a process:

1. Process Identification and Definition
2. Process Objectives and Criteria
3. Inputs and Outputs,
4. Strategies, Standards, Guidelines, and Rules (documented work practices and procedures) - the procedural basis for actions
5. Responsible Teams/Individuals (skills required),
6. Control Mechanisms (to evaluate Outputs and other elements from this checklist when applicable)- SQA background processes
7. Tools and Templates (assets)
8. Subject-Product and Quality Records-Database,
9. Approvals
10. Formal Process Improvement Procedures

The organization first specifies *what it does* through the Level I document (in reality a quality manual). In effect, Level I represents a statement of policy. Second level documents expand on this statement by providing the details (who, what, why, when, where) of operations and quality assurance procedures. In the third level the process definitions continue downstream by including the "how" manuals, and procedures, lists of instructions and definitions of practices. An organization is expected to comply with ISO requirements by doing what it says it does in its own quality documentation. It must record actions and be able to prove they were carried out by producing such documented evidence as plans, quality records, control records etc. The Fourth level is necessary in order to present records demonstrating this objective evidence.

What follows is the table of contents for the SDI diagnostic questionnaire. The complete set of contents may be found in the SDI Guide to Administration of the Level II Diagnostic Questionnaire (D'Artists Consulting Group, 1993). However, for the purpose of illustration we have embedded several illustrative questions in the text that follows.

These questions require proof that one of two (and only two) practical criteria have been met. The Unit must demonstrate that the process consistently (e.g., all sub-processes of the unit's software process satisfy applicable elements of the Level II Diagnostic checklist) satisfies detailed requirements as defined by our checklist (which itself is derived from the spirit of the ISO Quality Standards: ISO 9001, ISO 9000-3 and ISO 9004-2) Or, it must be able to demonstrate the documented and auditable

presence of a software process (a technology for the production of software) which fits the explicit definition of a quality element as specified in the appropriate ISO Standard.

Note: here technology is defined in one of two ways. It is either a reusable *template* for a model-prototype of the product, or it is an explicit operational definition of a process.

GI- Quality System Framework

a1- Management Representative
a2- Quality system; (Daily, 1992 is instructive)
a3- Internal Audits (Hollockamp, 1990 gives good example)
a4- Corrective actions

G2-Life Cycle Activities

a5- Contract Review

G2-a5-1-*A quality process is in place and in use* which will allow the Unit to understand and document customer requirements including nonfunctional requirements

G2-a5-2-*A quality process is in place and in use* for project-acceptance/risk- assessment

G2-a5-3-*Template and/or operational definitions are in place and in use incorporating:* acceptance criteria, systematic processes for management of changes in the customer's requirements during development, mechanisms for resolving variances with the customer in advance, methods for assigning responsibility for subcontracted work, methods for handling problems detected after acceptance

G2-a5-4-*Templates and/or operational definitions are in place and in use incorporating:* a procedure to establish, maintain, and document orderly contract reviews, as well as contract templates

G2-a5-5-*A quality process is in place and in use:* to verify that only functions the Unit is capable of handling will be contained in documents issuing out of the Unit; from the initial proposal to the final contract

a6- Development Planning
a7- Quality Planning
a8- Design and Implementation
a9- Testing and Verification

a10 Acceptance
a11 Replication, Delivery and Installation

a12 Maintenance

G2-a12-1-*Templates and/or operational definitions are in place and in use incorporating:* specifications of the scope of the maintenance activity and maintenance plans

G2-a12-2-*A quality process is in place and in use:* for the introduction and control of changes

G2-a12-3-*Templates and/or operational definitions are in place and in use incorporating:* resources for the maintenance activities identified by the developer and checklists for the definition of those resources

G2-a12-4-*A quality process is in place and in use:* for the development of Maintenance standards and dissemination of examples of good work

G2-a12-5-*A quality process is in place and in use:* for specifying maintenance record keeping and record documentation.

G3- Supporting activities

a13 Configuration Management

G2-a13-1-*Quality processes are in place and are in use:* for uniquely identifying each version of each software item; for identifying those versions of each software item which in combination make up a specific version of a complete product; for controlling the simultaneous updating of a given software item by more than one person; and (if indicated) for the coordination and updating of multiple copies of the same products in one or more locations.

G2-a13-2-*Templates and/or operational definitions are in place and in use incorporating:* procedures to identify and produce current build status of software products

G2-a13-3-*Templates and/or operational definitions are in place and in use incorporating:* guidelines specifying the circumstances under which an item should be brought under configuration control,

G2-a13-4*Quality processes are in place and are in use:* for the identification, documentation, review and authorization of changes under configuration management; for identifying activities resulting from a change request and for tracking them throughout its development; for notification of those

impacted showing the traceability between changes and the modified parts of the configuration items; for relating the configuration item back to the contract requirements; and for linking all documents and computer files related to configuration item.

G2-a13-5-*Templates and/or operational definitions are in place and in use:* for assuring that validation and confirmation of changes and identification of effects on other items are conducted prior to change acceptance.

a14 Document Control
a15 Quality Records
a16 Measurement
a17 Rules, Practices, and Conventions
a18 Purchasing
a19 Included (but not produced by the Unit) Sòftware Products
a20 Training

Procedures for Administering the Level II Questionnaire

There is a standard approach for administering the Level II Diagnostic Questionnaire. Each question is answered by every element of the Unit and than aggregated into averages per question, per area, per group and per Unit. The purpose of the second level is to:

a) diagnose less than satisfactory areas (specifically defined as less than four Yes answers),

b) identify processes that are the probable root causes of potential noncompliance.

Verification will be accomplished by cross checking individual answers to each item on the questionnaire with evidence developed using the checklist. Beside verification, expert analysis of this evidence, supplemented by clarifying interviews (one per noncompliance) will supply additional insight into possibilities for improvement providing a complete picture. In order to carry this out, a Software Process Model must be in place prior to execution of the diagnostic questionnaire. A template relevant to our quality concerns is represented by the Rolling Hexagonal - Verification Model (see figure one). The final product of the Level II Diagnostic Questionnaire for every Unit under assessment is:

1.) an executive summary of the Software Quality Program (SQPA-ES)

2.) **a report containing a detailed analysis of the Unit Status (SQPA-RR)**

3.) **a target diagram (figure two) with segments representing ISO requirements and concentric rings representing achievement levels.**

SDI Level III: The Improvement Level

The third level provides the basis for **Improvement.** Level III's standard set of questions are designed to measure the organization's current level of software process evolution relative to standard industry practices. It can be assumed that based on the **Diagnosis** phase we will first apply the third level questionnaire in identified areas of concern (e.g., those areas furthest out of compliance). It can also be assumed that different symptoms point to the same problem. Therefore, the output of the third level of the SDI system is a set of consistent recommendations which will create an orderly baseline on which an improved software process can be constructed. The logic of the third level questionnaire is implemented through an explicit decision model (e.g., a set of expert system rules represented by decision tables), which will match the answers to the questions with selected remedies. For every question five answers are given representing five successive levels of attainment. By selecting the answer that is closest to the situation for all the questions in designated areas, an entire set of remedies can be recommended.

Because the SDI model uses standardized, widely accepted industrial practices to form the baseline constant it is not static. It is really a generator of suggestions for quality programs because it is possible for it to evolve new practices as conditions in industry change, or when a piece of technology matures. Thus it provides a timely and comprehensive set of recommendations in response to changing conditions or differences in product line. Line software managers can quickly devise strategic and operational plans for software process quality improvement or necessary improvement to attain the compliance requirements built into ISO 9000-3.

Examples of Improvement Level III items referenced to the sample questions shown for Diagnostic Level II are as follows:

Contract (templates, incorporated practices and) reviews

1-A system to understand customer requirements is in place, that covers
  a) Application level only
  b) System level only

Areas of ISO 9000-3 requirements
by SDI Questionnarie Level II (1-20)

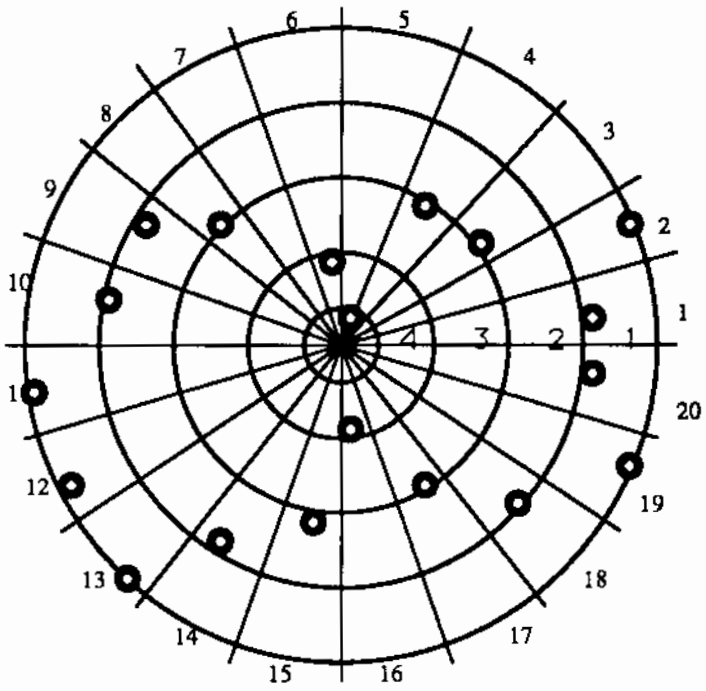


Figure two. Target Diagram

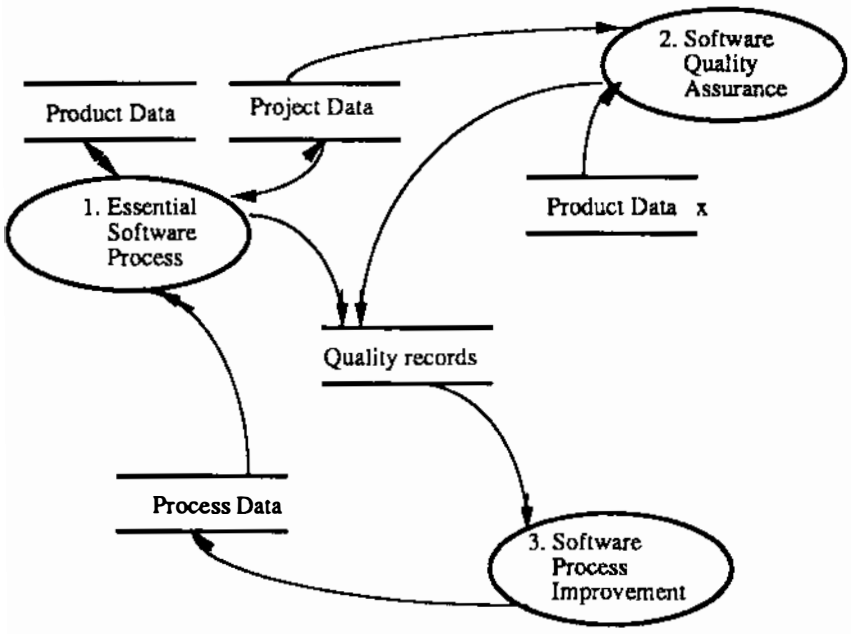


Figure three. Coordination of Software Processes

c) Network level and Application level
d) System and Application level
e) System, Network and Application level requirements

2-A clear approach to project-acceptance exists that covers:
a) cost assessment
b) risk assessment
c) cost assessment and risk assessment
d) technology selection based on risk assessment
e) value for the customer, cost and capability for the provider and project-product technology selection based on risk assessment

3- Templates exist for:
a) proposal formulation,
b) contract including changes
c) proposal formulation, contract and project-product architecture
d) proposal formulation and nonfunctional-quality requirements
e) proposal, contract including changes, acceptance criteria, subcontractors, project-product architecture and nonfunctional-quality requirements

It makes sense to assume that most organizations would want ongoing and continuous improvement of their software process, notwithstanding the need to meet identified requirements for ISO compliance. Toward that end, the following list presents a set of aggregated approaches that establish either data/process/or communication remedies. For example, depending on the results of the Level III Process Improvement Questionnaire, it might be recommended to establish:

Data
A Teamwork Support Database
A Measurement Database
A Database of Standards
A Quality Records Database
A Reusable Assets Database (Reusable Libraries)

Process
Planning and Development Tools
Standard Methodologies
Training (for example in Design and Maintenance Reviews)
Software Process Intelligence System

Access
Enhanced Access to Supplier Information and Personnel
Enhanced Access to Customer Information and Personnel
Enhanced Access to Management/Professionals in the Software Field

Conclusion

This paper presents the SDI (**Sense-Diagnose-Improve**) model for the internal audit of ISO 9000-3 compliance. The finished model evolved out of a three year pilot study, in which over twenty prototype questionnaires were developed and administered by management professionals from more than 100 organizations in the Detroit metropolitan area. The theoretical basis comes from twenty years of practical, industry experience and university research. The crystallization of this work was brought about by the promulgation of the ISO 9000-3 Standard (1991).

SDI is designed to clarify ISO 9000 series requirements as they apply to software production, specifically for business oriented software development and maintenance. SDI provides a set of consistent recommendations which will create an orderly baseline on which an improved software process can be constructed. The SDI model provides a timely and comprehensive set of recommendations in response to changing conditions or differences in product line.

Software managers can quickly devise strategic and operational plans for software process quality improvement, or to attain the compliance requirements built into ISO 9000-3. Level one is a prerequisite assessment of upper management commitment. During this phase the importance and consequences of a quality system is characterized for upper management and their participation is obtained. In Level two a detailed diagnostic questionnaire is administered aimed at determining areas where quality must be improved in order to achieve compliance with ISO 9001 and ISO 9000-3. Conceptually level two is determined by the standards themselves. Level three is a vehicle to govern preparation of software process improvement recommendations. It establishes orderly baselines in compliance with the ISO 9000-3 and accepted practices of the software industry.

References

1. L.J. Artur "Improving Software Quality- An Insider's Guide to TQM" J. Wiley 1993,
2. J.Baumert"New SEI Maturity Model Targets Key Practices"IEEE Software November 1991,
3 "A.Bisego, at al. "Bootstrap:Europe's Assessment Method" IEEE Software May 1993,
4. T.Bollinger, C.McGowan "A Critical Look at Software Capability Evaluations" IEEE Software July 1991,
5. W. Carlson ,McNurlin "Measuring the value of Information System" Special IS Analyzer report UCG 1989,
6. Carlson, McNurlin "Uncovering The Information Technology Payoffs" UCG IS Analyzer Special report 1992,
7. P.Crosby "Quality is Free" McGrow Hill 1979,
8. K.Daily " Quality Software Management" NCC Blackwell, Oxford 1992,
9. R.Grady D.Caswell "Software Metrics Establishing a company wide program"Prentice hall 1987,
10. R.Grady "Practical Software Metrics for project management and process Improvement" Prentice Hall 1992,
11. Hetzel "The Complete Guide to Software Testing" 2ed. QED 1988,
12. C.Hollocker "Software Reviews and Audits Handbook" J.Wiley 1990,
13. W.Humphry, ... "A Method for Assessing the Software Engineering Capability of Contractors" Technical Report CMU/SEI-87-TR-23 and ESD/TR-87186, September 1987,
14. W. Humphry "Software Process" Addison Wesley 1988,
15. W. Humphry "Comments on a Critical Look", IEEE Software July 1991,
16. "IEEE Guide for Software Quality Assurance Planning", ANSI/IEEE, 1986,
17. ISO 9001 "Model for Quality Assurance in Design/Development, Production Installation and Servicing" 1987,
18. ISO 9004-2 "Quality management and quality system elements", 1991,
19. ISO 9000-3 Guidelines for the implementation of ISO 9001 for the development of software" 1991,
20. "ISO 9000: Handbook of Quality Standards and Compliance" BBP 1992,
21. C. Jones "Programming Productivity" McGrow Hill 1986,
22. C.Jones "Applied Software Measurement - Assuring Productivity and Quality" McGrow Hill 1991,
23. J. Lamprecht "ISO 9000 preparing for registration" Marcel Dekker 1992,

24. J. Marchiniak , D.Reifer "Software Acquisition Management" J.Wiley 1990,
25.. M.Ould "Strategies for Software Engineering- The Management of Risk and Quality" J.Wiley 1990,
26. M. Parker, R.Benson, H.Trenor "Information Economics", Prentice-Hall, 1988,
27. W. Perry, "Quality Assurance for Information Systems (Methods, Tools, and Techniques)", QED Technical Publishing Group, Boston, 1991,
28 H. Rubin "Measure for measure" Computerworld April 1991,
29. M. Steeples "The Corporate Guide to Malcolm Baldridge National Quality Award" second edition, Irwin 1993,
30. G. Shulmayer "Handbook of Software Quality Assurance" Van Nostrand 1992.
31. T.Vollman "Software Quality Assessment and Standards" Computer, June 1993.
32. M.Paulk, B.Curtis, M.Chrissis, C.Weber "Capability Maturity Model, Version 1.1" IEEE Software, July 1993.

Process vs. Product Assurance: Is There Really a Conflict?

John Joseph Chilenski
Boeing Commercial Airplane Group
PO Box 3707, MS-6Y-07
Seattle, WA 98124-2207

Abstract

In most engineering fields, process and product assurance go hand in hand. Reasons why software must consider both simultaneously are discussed.

Presenter

John Joseph Chilenski is a Principal Engineer at Boeing Commercial Airplane Group in the Avionics/Flight Systems Software Engineering Technology organization. He holds a BS in Electrical Engineering and Computer Science and Nuclear Engineering from the University of California at Berkeley. He is Principal Investigator for Software Verification and Validation Research.

The Role of the Quality Group in Software Development

Douglas Hoffman
Software Quality Methods
1449 San Tomas Aquino Road
San Jose, CA 95130
(408) 984-7560

Abstract

This paper describes the role of the quality organization in software development as observed in dozens of commercial organizations. It views the different charters and purposes the quality groups have. The potential benefits and drawbacks for various charters are presented, along with the organizational structure and typical activities in each. The charter for the quality group changes over time, and observations of progressions in organizations are made. The various possible organizations, charters, and roles are described and related briefly to quality systems described in both the SEI Capability Maturity Model and ISO 9000 Standards (ISO 9001 and ISO 9000-3). The impact on product quality of the different types of development process, and possible roles for the quality group are also covered.

About the Author

Mr. Douglas Hoffman has been in the software engineering and quality assurance fields for over 20 years. He is currently an independent consultant with Software Quality Methods, and specializes in identifying the appropriate development processes and tools for software quality based upon organizational requirements. Currently, he is Chairman of the Santa Clara Valley Software Quality Association (SSQA), a Task Group of the American Society for Quality Control (ASQC), and Program Chairman for the Third International Conference for Software Quality. He is also active in the local section of the ASQC and the ISO 9000 Task Group, and is registered as an ISO 9000 Provisional Auditor. He received his MBA from Santa Clara University, and his MS in Electrical Engineering and BA in Computer Science from UC Santa Barbara.

The Role of the Quality Group in Software Development

I. Introduction

The purpose, or mission, of software quality organizations ranges from testing products to providing information and expertise about the product and development process. The group may also provide knowledge and training on product testing, process creation and management, toolsets, and metrics. Various tasks for the software quality group are covered, and the order in which they typically appear as the organizations grow and mature is outlined. This provides a foundation for understanding the value of the quality assurance organization and the contribution they make to the product and process quality.

The roles of software quality assurance can be described with the tasks they undertake. The roles range from acting as an extension of development for debugging software products, to development process definition and control. Verification and validation, acceptance testing, measurement and metrics, and process consulting are also roles that software quality groups sometimes assume. The various charters that the organization may assume are described, and the impact on quality is addressed for each charter.

As the organizations grow and change, the needs and roles also change. Depending on the type of product and organization itself, the life cycles may differ and the tasks done by the quality organization evolve. The evolution takes familiar tracks, following patterns based upon the maturity of the organization and other factors. The SEI Maturity Model and other standards are relevant in understanding the importance and roles for the quality group.

II. Quality Group Purposes and Roles

Figure 1 shows the basic purposes, roles, and activities established for software quality groups. Although an organization may fit the description well at one time, it is likely to change as the organization evolves and matures. It is also likely that any given quality group has characteristics of many of the organizations. Generally, however, there is one primary or predominant theme in the group.

When the goal of the organization is to test products, the group usually acts as an extension of the development organization in performing a debugging function. This is shown as the Type 1 row in Figure 1. The group's primary activities are to develop and run tests, with the primary emphasis on reporting defects. The majority of time is generally spent running the tests and reporting the results. This role gives the most tangible and obvious results, yet is the least leveraged and thus the least productive. Most organizations encountered in industry begin with this goal, and often even well established groups are relegated to being only testers.

This first type of group is doing quality control; acting as a filter for products to ensure that only products with acceptable levels of errors are delivered. The activities are reactive,

responding to past events, and the only real power in the organization has is in stopping products from being delivered. In the most extreme case, the group is acting as a debugging function for development, with little or no effect on the cause of quality in the software. The functioning of the group as quality assurance doesn't begin until the group is operating as a type 3, 4, or 5.

Another, more subtle version of the testing goal is to measure the quality of products. this is shown as Type 2 in the table. This differs from simply testing in the group's arms-length relationship with the testing and the data, and emphasis within the organization on achievement of specific levels of quality. The quality group often does not do the primary testing, but rather oversees and reports on the results of other groups' activities. Although they may act in a more advisory capacity, the focus in Type 2 is still on testing and measuring the quality through product defects. Both forms of organizations have been observed, performing arms-length tests and recording the results from other groups' testing.

Often the Type 2 organization is gathering enough data on the product that good quantitative decisions and predictions can be made. This sometimes results in the group functioning as a "quality hurdle"; a quality level to be achieved before the product can be released. Depending on how the acceptable quality levels are determined, and the way the group interacts with other organizations, this can be a large step forward from a Type 1 group.

A different focus for the quality group occurs when they concentrate more on the organization's processes, rather than the products. This often occurs when the organization focuses on metrics programs and expands beyond the nebulous "defects happen" theory into an understanding that "defects are built in". This is the Type 3 organization. The role of the quality group becomes more general, that of information brokers seeking insights from whatever data is obtainable. Metrics programs change the role of the quality group to information engineers; applying the data to understand and improve the organization. Few quality organizations have progressed to using sophisticated metrics, and even fewer have been successful in applying them to predict program and process behavior.

Armed with the information provided by the metrics, the quality group can begin to assure that the software is of good quality. Even without planning the processes, the knowledge of what works and what doesn't is powerful enough to change what is done. This information also is the foundation of quality improvement programs. The group is not purely reactive at this level.

Type	Purposes	Activities	Roles
1	Test Products	Test development, test execution	Testers; extension of development
2	Measure Products	Test oversight, reporting results	Measurers; Quality hurdle
3	Measure Processes	Metrics	Information Engineers
4	Define Processes	Process and Risk management	Quality and Process Engineers
5	Guidance Resource	Quality Reference	Quality Engineers

Figure 1: Types of Quality Groups

When process definition is the goal, the quality organization performs more of a role of quality and process engineering. They assess risks and design processes to reduce the risks and increase the quality. This becomes more of a management approach than a technical one. Organizations often form a separate group for process engineering, but this is different from the quality group concentrating on ensuring that the process is properly defined, measured, and controlled.

As a guidance resource, the quality group provides expertise and reference information so others in the organization can effectively do their jobs and improve quality. This reference information includes how-to as well as what-to guides, and measures of quality and their meanings. This is more than arms-length measurement and reporting, but provides guidance, training, and assistance in issues relating to quality throughout the organization.

III. Software Development Life Cycles

Several Software Development Life Cycles (SDLC) are described in Figure 2, and situations where they are most applicable and effective are shown. The SDLCs used in organizations include those shown and variations created to address specific organizations' needs. The appropriateness for the life cycles is shown in relation to the stability and understanding of product requirements. It is these requirements and their characteristics that appear to be the underlying driver of life cycle selection.

For example, the classic waterfall approach to software development is most appropriate when the requirements can be fully known before beginning development, and they don't change substantially during the product development. If they change substantially, a spiral approach is more likely to fit the organization's needs. Variations are usually developed for customization to the unique organizational requirements. Variations are required for situations such as when the requirements are initially known but subject to change.

Product Requirements	Life Cycle
Known, unchanging	Waterfall
Unknown, changing	Prototyping
Unknown, unchanging	Spiral
Known, unchanging	Decomposition/Integration
Known, provable	Cleanroom
Unknown	Fourth Generation Techniques

Figure 2: Most Appropriate Software Development Life Cycles Based Upon Product Requirements

The role of the quality group is independent of the life cycle involved. The role is primarily a function of the maturity of the organization and the appropriateness of the life cycle model. The specific activities differ on a technical level, but the various possible roles remain the same, and the progression and evolution usually occur in the same ways. Problems occur in an organization when the wrong life cycle is chosen for a project, and in these instances the quality of the product is generally low, and the role of the quality group is relegated to testing.

The incorrect choice of life cycle causes wasted effort and frustration among the project members. It also results in lower quality for the product. For example, changing requirements frequently when using a waterfall approach causes respecification, redesign, recoding, and retesting. The fixes to accommodate the changes cause the quality to drop and the code to age quickly. In these situations, it is also impossible to properly plan for the testing. This means continual rewriting of test cases and rerunning of the test suites. The quality group does not have time to properly plan and create stable measures of quality. Similar problems occur whenever a poor fit is chosen for the life cycle approach.

IV. Organization Growth and Maturity

Organization maturity is not an indication of the age of the group. It can be defined as a measure of the formality of the processes used by software development. For healthy organizations, the role slowly evolves through the maturity levels. The level of maturity of the organization roughly correlates with the role of the software quality group.

The relationship of the software quality assurance group role to SEI's Capability Maturity Model (CMM) is shown in Figure 3. For each level of maturity, the roles for the software quality group is shown. The majority of organizations are at level 1 or 2, with a few delving into metrics and process and risk management.

The relationship between the organization maturity and the role of the software quality group is worth understanding. Although they seem closely correlated, I believe there is a chicken-and-egg problem in trying to determine which causes which. The more mature the group, the more the responsibility for quality is distributed throughout the organization.

The role of the quality group evolves from testing to process definition and control as the organization evolves. Trying to control and optimize the development process in an organization at the Initial Level does not make sense. The process at the Initial level is poorly defined and inconsistent. Control and optimization in this situation is not definable; there is no way to know a process is controlled if one does not know what the process is. On the other hand, paying no attention to process does not make sense either. A balance must be struck between definition and control of process now and improvement of the process for the future.

The most effective role for the quality group is the one that best supports the organization today, while preparing to improve it in the near term. Without necessarily advocating any particular methodology for organization development, the quality group must understand and support some SDLC model - what ever model the organization agrees fits its needs. When the appropriate SDLC is applied, development goes smoothly and good quality practices can be more

easily applied. When a poor choice is made of the SDLC (or no choice is made), the effort by all members of the organization is increased and the quality decreased. This forces the quality group to act as testers and gate keepers to avoid release of seriously faulty products.

SEI Maturity Level	Role of Software Quality Assurance
1- Initial	Testing
2- Repeatable	Quality hurdle
3- Defined	Oversight, Metrics
4- Managed	Process and Risk management
5- Optimizing	Reference, Oversight

Figure 3: Organization Maturity and SQA Roles

V. Models for the Quality System

Other models and standards for quality organizations and organizational evolution are also applicable. ISO 9000 is a framework for a quality system, rather than a process methodology or prescription for the software quality organization's charter or function. The relationship of the quality system to the business system and development methodology is graphically described in Figure 4. The quality organization charter in this model can be as controller of the process or the product.

The quality framework described by ISO 9000 is one which sits on top of, and becomes part of the normal business systems. It is directed at the production process rather than the product directly. In the case of software development, the business system is the development methodology and support mechanisms. The quality system is the set of monitors and controls applied to the business system to ensure that it is working, rather than testing of the product itself to see if it works.

Neither SEI's Capability Maturity Model nor ISO 9000 describe in detail what the right process is, who should do what, or how things should be done. ISO does not begin to prescribe these things, but rather provides rules for knowing if a given quality system might qualify under its guidelines. Neither system addresses product quality directly.

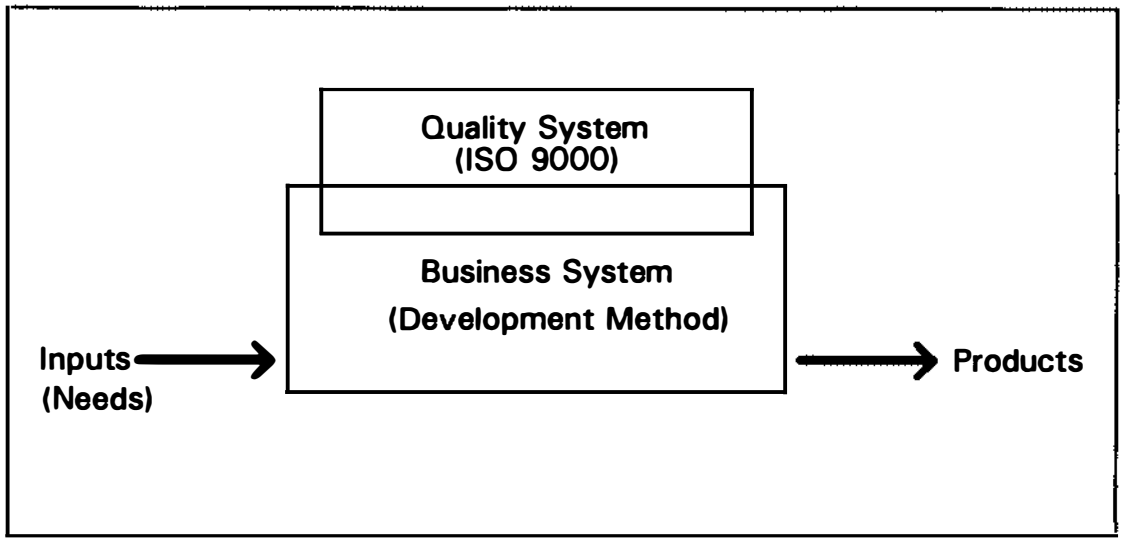


Figure 4: Quality System and Business System Relationship

The models don't prescribe specific techniques or methods because each organization's situation is so different that there is no single solution. In order for generalized models for development organizations or quality systems to be useful, they must be applicable in many situations. If these models prescribed specific methods and techniques, they would not be applicable to the majority of organizations since they have such different needs and characteristics. Organizational requirements are unique, and are based upon such things as the product characteristics, technical development environment, customer expectations, and organizational politics.

The models are also process oriented, not product oriented. They focus on the processes organizations should have, not the products. They do not directly address testing of products or product quality. They describe how the process must be defined, controlled, and improved. Only by controlling the process can the product quality be predicted and controlled.

VI. Continuous Improvement

There are a number of factors that must be in place before a continuous improvement program can succeed. A system must be defined and in place to be improved. Improvement is a relative concept that assumes some knowledge of the quality before and after some event, therefore measurement is required. Continuous improvement requires continuous measurement and comparison to ensure that the processes are under control and the changes implemented do indeed improve things. As described in the CMM, software development organizations usually evolve from less structured to more formally functioning. These changes take months or years, and generally must be planned and nurtured to be successful.

At the lowest level of structure, there is not enough defined process to measure and compare. When an organization has matured beyond trial-and-error approaches to software development, it

is possible to begin to formally monitor and improve the processes. The quality group can be instrumental for both monitoring and defining the processes. The monitoring takes place both on the process directly and on the products to confirm indirectly that the processes are under control. In mature organizations, measurement of product quality is necessary and routine to ensure that the processes are under control and to monitor the outcomes of changes implemented for improvement.

VII. Conclusion

The quality group's role travels full circle as the quality system takes root. When the system is beginning, or the organization is in trouble, the quality group concentrates on testing of product and acting as gate keepers. As the quality system becomes established, emphasis shifts toward process and measures. Intervention is often necessary, and the quality group may take an active role in implementing change. Once the quality management system is well established, only small amounts of attention are needed to monitor and review activities to ensure that the processes remain under control. With a well established quality system, the quality group role moves back more to arms-length measuring and reporting, but with much higher leverage and effect.

How can this be applied? First, we need to establish what the organization is doing. The role of the quality group should be set based upon the needs of the organization. These needs can be predicted by the maturity of the organization and the need to change. The appropriateness of the SDLC can be evaluated and changes made if required. Then goals for improvement of the process and evolution of the organization can be set. The quality group can play a big part in the planning and implementation through understanding of organizational development needs and techniques. Then an improvement program to attain the goals can be begun. This is the foundation of any continuous improvement program, and ultimately should be the goal of the software quality group and all of management in the organization.

References

Humphrey, W.S., *Managing the Software Process,* Addison-Wesley, 1989

Humphrey, W., *"Characterizing the Software Process: A Maturity Framework,"* IEEE Software (March), 73-79, 1988.

Quality Management and Quality Assurance Standards, ASQC, documents ANSI/ASQC Q90, Q91, Q92, Q93, and Q94-1987. (ISO 9000)

Experiences with CSRS: An Instrumented Software Review Environment

Philip M. Johnson
Danu Tjahjono
Dadong Wan
Robert S. Brewer

Department of Information and Computer Sciences
University of Hawaii
2565 The Mall
Honolulu, HI 96822 U.S.A.

Abstract

Formal technical review (FTR) is a cornerstone of software quality assurance. However, the labor-intensive and manual nature of review, along with basic unresolved questions about its process and products, means that review is typically under-utilized or inefficiently applied within the software development process. This paper discusses our initial experiments using CSRS, an instrumented, computer-supported cooperative work environment for software review that reduces the manual, labor-intensive nature of review activities and supports quantitative study of the process and products of review. Our results indicate that CSRS increases both the breadth and depth of information captured per person-hour of review time, and that its design captures interesting measures of review process, products, and effort.

Biographical Information

Philip Johnson is an Assistant Professor of Computer and Information Sciences at the University of Hawaii. Dr. Johnson is Director of the Collaborative Software Development Laboratory (CSDL), which performs research on tools and techniques for group-based software engineering, software quality assurance, and other collaborative activities. Danu Tjahjono and Dadong Wan are members of CSDL and doctoral candidates in the Communication and Information Sciences program at the University of Hawaii. Robert Brewer is a member of CSDL and a graduate student in the Computer and Information Sciences Department at the University of Hawaii. Address e-mail correspondence to `Johnson@Hawaii.Edu`.

1. Introduction

Formal technical review (FTR) is a cornerstone of software quality assurance. While other techniques, such as software testing can help assess or improve quality, they cannot supplant the benefits achievable from well-executed FTR. One reason why review is essentially irreplacable is because it can be carried out early in the development process, well before formal artifacts such as source code are available for complexity analysis or testing. A more important reason is because no automated process can yet provide the two-way quality improvement in both product and producers possible through review.

However, the full potential of review is rarely realized in its current forms. Three significant roadblocks to fully effective review are the following:

Review is extremely labor-intensive. Typical procedures for FTR involve individual study of hard-copy designs or source listings and the creation of hand-generated annotations, followed by a group meeting where the documents are paraphrased line by line, issues are individually raised, discussed, and recorded by hand, leading eventually to rework assignments and resulting changes. For one approach to FTR called *code inspection* (Fagan, 1976), published data indicates that an entire person-year of effort is needed to review a 20 KLOC program by a team of four reviewers (Russel, 1991). Unfortunately, little automated support for the process and products of review is available. What support is available typically involves only a single facet of review (such as the review meeting), or is not integrated with the overall development environment.

Review is not compatible with incremental development methods. Because of their labor-intensive nature, most organizations cannot afford to review most or even many of the artifacts produced during development. Instead, review is deployed as a "hurdle" to be jumped a small number of times at strategic points during development. While this may be a reasonable tactic for development in accordance with a strict waterfall lifecycle model, more modern incremental and maintenance-intensive development methods prove problematic: there is no effective way to optimally position a small number of review hurdles in the development process.

No methods or tools exist to support the design of prescriptive review methods adapted to an organization's own culture, application area, and quality requirements. Research on review tends to fall into two categories, which we will term "descriptive" and "prescriptive". The descriptive literature describes the process and products of review abstractly, advocates that organizations must create their own individualized form of review, but provides little prescriptive support for this process (Schulmeyer, 1987; Dunn, 1990; Freedman, 1990). Such work leaves ill-defined many central questions concerning review, such as: How much should be reviewed at one time? What issues should be raised during review, and are standard issue lists effective? What is the relationship between time spent in various review activities and its productivity? How many people should be involved in a review? What artifacts should be produced and consumed during a review? The prescriptive literature, on the other hand, takes a relatively hard line stance on both the process and products of review (Fagan, 1976; Fagan, 1986; Russel, 1991). Such literature makes clear statements about the process (Meetings must last a maximum of 2 hours; each line of code must be paraphrased; lines of code must be read at rate of 150 lines per hour; etc). The data presented in

this literature certainly supports the claim that this method, if followed precisely, can discover errors. However, the strict prescriptions appear to suggest that organizations must adapt to the review method, rather than that the review method adapt to the needs and characteristics of the organization.

This paper describes our initial experiences using CSRS[1], a computer-supported cooperative work system that is designed to address aspects of each of these three roadblocks to effective formal technical review.

First, CSRS is implemented on top of EGRET, a multi-user, distributed, hypertext-based collaborative environment (Johnson, 1992) that provides computational support for the process and products of review and inspection. EGRET runs in a Unix/X window environment, providing a client-server architecture with a custom hypertext database server back-end (Wiil, 1990) connected over an ethernet network to front-end EMACS-based clients. This platform allows an essentially "paperless" approach to review, supports important computational services, and facilitates integration with existing development environments. Our initial experiments indicate that CSRS effectively eliminates many of the manual, clerical tasks associated with traditional review. In addition, our experiments show that CSRS captures significantly more information during review than traditional, manual approaches.

Second, CSRS is designed to interoperate with an incremental model of software development. While simply lowering the cost helps integrate review into incremental models, CSRS additionally provides an intrinsically cyclical process model that parallels the iterative nature of incremental development. Our initial experiments also indicate that CSRS supports interoperation by capturing significant amounts of information relevent to other development phases, such as design rationale information.

Third, CSRS exploits the use of an on-line, collaborative environment for review with instrumentation designed to collect a wide range of metrics on the process and products of review. Such metrics provide historical data about review process and products for a given organization, application, and review group that can provide quantitative answers to many of the questions concerning the basic parameters for review raised above. Our initial experiments indicate that CSRS can provide novel data to organizations at a far smaller grain-size than traditional measurement approaches.

CSRS represents two major paradigm shifts with respect to traditional review methods. First, CSRS changes review from an essentially "off-line" process to an essentially "on-line" process[2]. Second, CSRS instrumentation collects measurements at a very small grain-size. (For example, CSRS automatically collects data such as the minutes spent per reviewer on individual functions, as opposed to traditional approaches where reviewers manually record the total elapsed time spent doing preparation, participating in the meeting, and so forth.) Systems involving paradigm shifts are inherently exploratory in nature, requiring an iterative process of design and evaluation to expose and resolve the issues that arise when making paradigmatic change. This paper describes the successes and failures of our experiences with CSRS to date, with the goal of facilitating other research in the design of review support systems.

---

[1] An acronym for Collaborative Software Review System.

[2] Other researchers currently investigating this paradigm shift are (Brothers, 1990), and (Gintell, 1993).

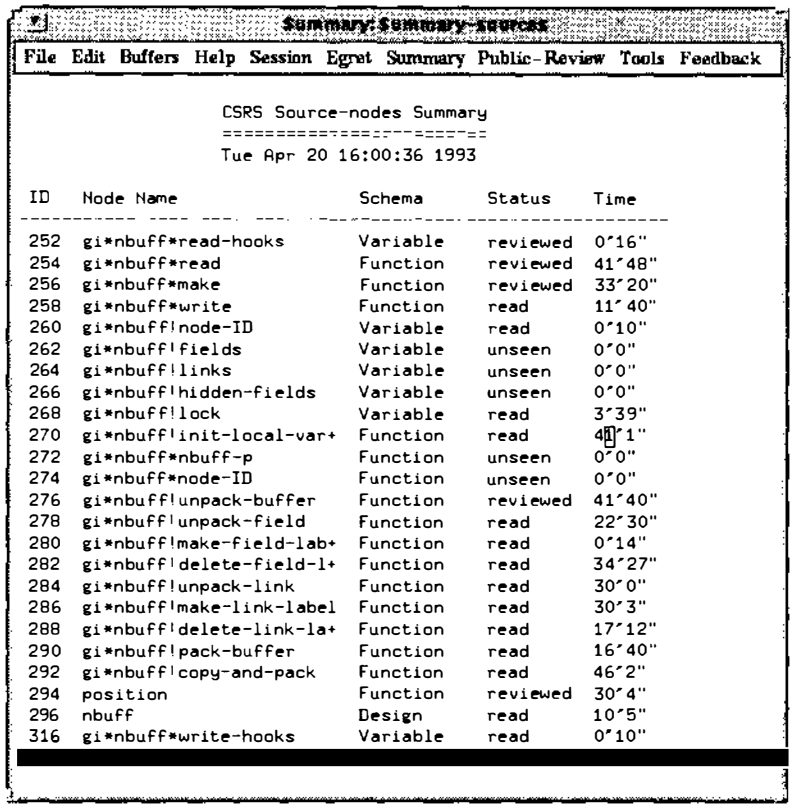


Figure 1. A summary window illustrating the state of review for one reviewer.

To orient the reader to on-line review, the next section provides a brief introduction to CSRS from a user-level perspective. The following sections discuss the design evolution of CSRS over the past year, detailing our experimental usage of CSRS, the implications of these experiences, and the current status of the system.

2. A review scenario using CSRS

To get the flavor of FTR using CSRS, this section presents excerpts from a recent review. This review cycle involved an object-oriented class implementation called "Nbuff" (short for node-buffer) in the generic-interface subsystem of EGRET. Nbuff defines an abstraction that bridges and combines the hypertext "node" abstraction provided by lower-level subsystems in EGRET and the textual "buffer" abstraction provided by EMACS and higher, application-specific subystems such as those comprising CSRS.

In CSRS, each program object, such as a function, procedure, macro, variable or data type declaration is retrieved from a source code control system and placed into its own node in a hypertext-style database. After an *orientation* session to familiarize each review participant with the system under study, a *private review* phase begins. During private review, each member individually reviews the source code without access to the review commentary of others, although non-evaluative questions and answers about requirements and so forth are publically accessable. CSRS provides facilities to summarize the state of review for the reviewer, such as the window

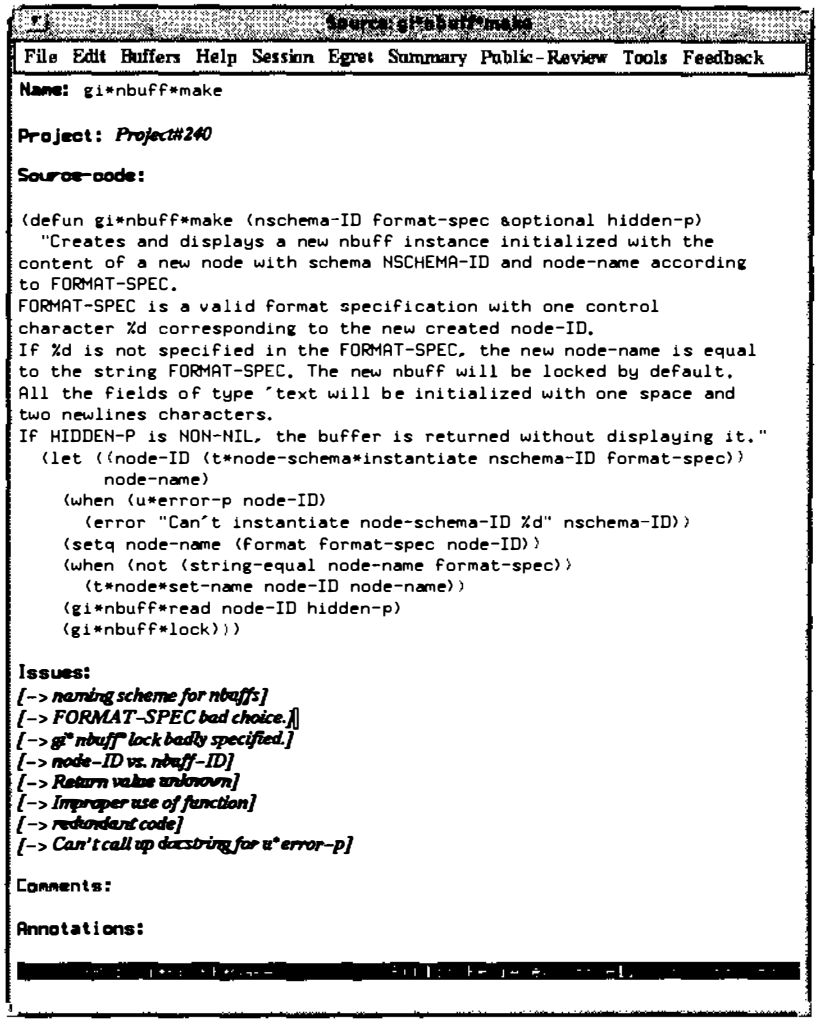


Figure 2. A source node illustrating one of the functions under review.

displayed in Figure 1. At this point, the reviewer has partially completed private review, as indicated by the fact that some of the source-nodes are reviewed, some have been read but have not been completely reviewed, and some have not yet even been seen.

By mouse-clicking on a line or through menu operations, the reviewer can traverse the hypertext network from this screen to a node containing a source object under review, as illustrated in Figure 2. In this case, the object under review is the operation gi*nbuff*make. Both pull-down and pop-up menus facilitate execution of the most common operations during this phase, such as creating an issue concerning the current source node under review (as illustrated in Figure 3), or proposing a specific action to address an issue. Once the reviewer is finished with a source object, he explicitly marks the node as "reviewed". Since this is the private review phase, only the issues created by this reviewer are accessable.

CSRS assumes that typical programming environment tools are available to the reviewer, such as static cross-referencing and dynamic behavior information, and thus does not attempt to duplicate that functionality. Part of the benefit of an EMACS-based platform is ease of integration with external programming environment tools (for example, EMACS interfaces are provided in the

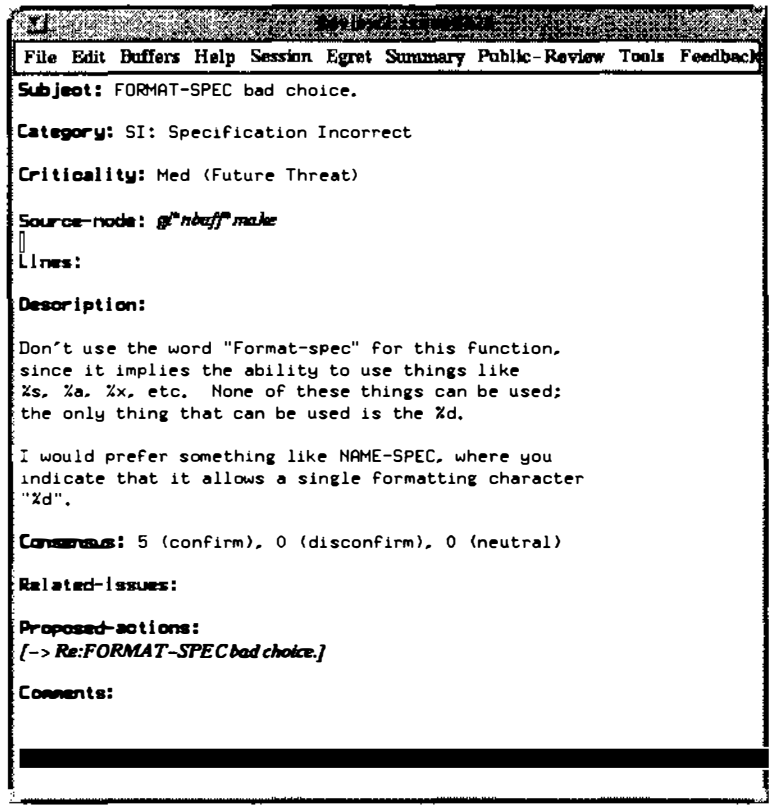


Figure 3. An issue node containing an objection to an aspect of gi*nbuff*make.

C/C++ environments XOBJECTCENTER and ENERGIZE, as well as in Common Lisp environments by Lucid and Franz.)

Once the source nodes have been privately reviewed, the *public review* phase begins, where reviewers now read and react to the issues and actions raised by others. Each reviewer responds to the issues and actions raised by others through the creation of new issues or actions, creation of confirming or disconfirming evidence nodes to extant issues or actions, and by voting for one or more actions to be taken during the rework phase.

Following public review, the moderator uses CSRS to *consolidate* the review state. Consolidation involves the restructuring of information captured during review into a written report that delineates the proposed actions, agreements, and unresolved issues resulting from the private and public review phases. CSRS provides automated support to the moderator in traversing the hypertext database and generating a LaTeX document containing the consolidated report.

If all issues arising from the on-line phases have been satisfactorily resolved, then this consolidation report is also the final review report that both specifies the issues raised and the rework required. Consolidation reports are far more comprehensive and detailed than typical review reports from traditional review methods, such as those described in (Pressman, 1992; Freedman and Weinberg, 1990; Humphrey, 1989). If the consolidation report reveals issues unresolved during public review, then the moderator schedules a face-to-face meeting to resolve these issues, or else decides them unilaterally. (CSRS is not currently used in

face-to-face review meetings. Lack of automated support for this phase of CSRS-style review has not been a problem in practice, since such meetings are typically very short or avoided altogether.)

(Johnson and Tjahjono, 1993) provides a detailed description of the data and process model used in CSRS. The next section reports on our experiences in the use of CSRS, and provides rationales and results from our design decisions.

3. Initial experiences with CSRS

Our research project on computer-supported FTR will be two years old at the time of publication, and we will have been performing review experiments with running versions of CSRS for over one year of that period. CSRS has evolved and matured significantly during this time. To most clearly present the successes and failures of CSRS to date, we will present the major releases of CSRS, interesting facets of their design rationale, and our experimental data in chronological order. As the next sections will reveal, we first focussed on the design of a data and process model appropriate for on-line review, then incrementally explored the design space for automated review instrumentation.

3.1 CSRS I: Process/data model design

In the first release of CSRS, we concentrated upon the impact that specializing our computer-supported cooperative work environment, EGRET (Johnson, 1992), to FTR would have upon the process and products of review. It became clear that the "classic" FTR method–Fagan-style code inspection (Fagan, 1976; Fagan, 1986)–cannot be straightforwardly applied to a collaborative work environment.

First, the manual nature of Fagan-style inspection means that certain concepts, such as the role of scribe, do not make sense in an automated environment where each participant's actions are captured automatically.

Second, since EGRET is oriented toward asynchronous communication, our support for review is similarly biased. However, the primary focus of Fagan-style inspection is on the nature of the synchronous, face-to-face meeting and its attendent issues. A change in process orientation from synchronous to asynchronous has profound implications. First, it leads to extensive change in the role of moderator. While the primary responsibilities of a Fagan-style moderator is to ensure reviewer preparedness and maintain order and effectiveness during a face-to-face meeting, a CSRS moderator's task involves two completely different issues: creating a well-structured hypertext database of source artifacts, and maintaining order and effectiveness while participants asynchronously read source code and other postings and reply to them on-line.

The change from synchronous to asynchronous interaction also allows change in the scope and content of review. One of the fundamental guidelines for effective synchronous review is "raise issues, don't resolve them." In other words, in a face-to-face meeting, it is important to keep focussed directly on the generation and recording of issues, and to avoid discussion of resolutions. Such a focus is needed because review meetings may cost 4-6 person-hours of skilled technical staff per hour of elapsed time, and resolution conversations are frequently not only time consuming, but may involve only a small subset of those attending.

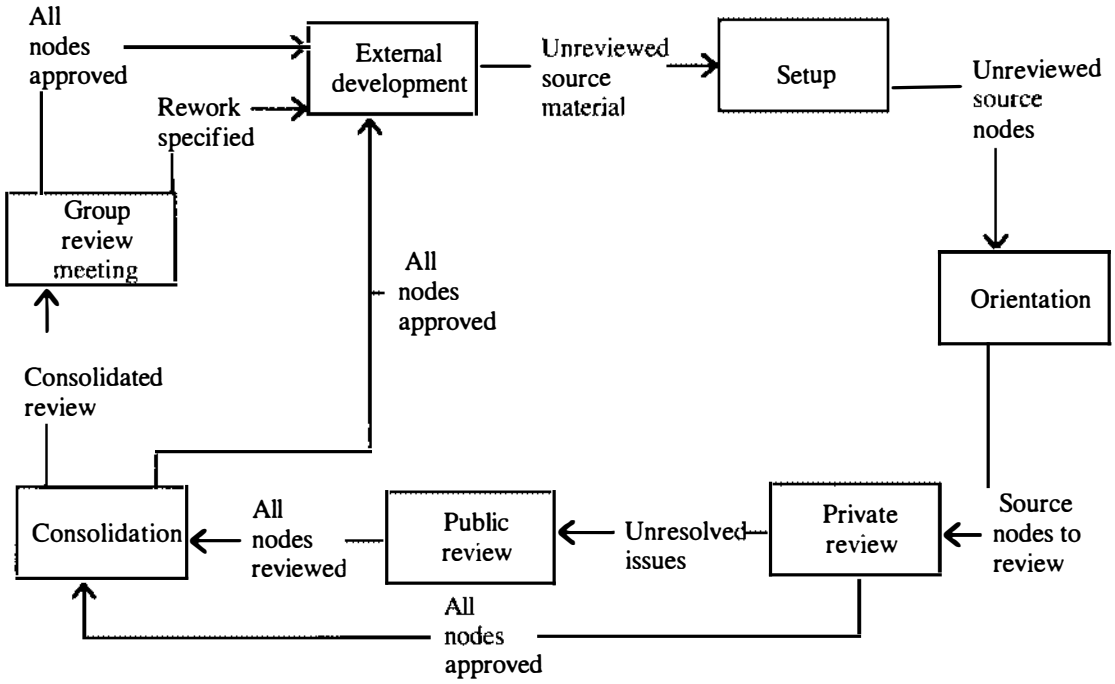


Figure 4. The CSRS process model. During the Setup phase, the Moderator and Producer use tools to build a hypertext database of source material. During Orientation, the producer provides the Reviewers with a high-level overview of the review materials. During Private Review, reviewers analyze source material and generate issue and action proposals. During Public Review, reviewers analyze the issues and actions generated by other reviewers, and attempt to build consensus. During Consolidation, the Moderator analyzes the current state of review. If all issues have been resolved in a consensual manner, then the review session terminates and a final report is generated. If controversies remain, a Group Review Meeting is called to resolve the issues in a face-to-face manner, followed by termination of review and generation of the final report.

Avoiding resolution-oriented discussion significantly improves the group process by preventing conversational digression and by improving the usage of human resources.

In an asynchronous environment, however, this rationale for restricting the scope of review no longer applies. Since participants are working asynchronously, time spent by one participant generating a potential resolution to an issue does not incur cost to others. During public review, only those participants qualified to evaluate an issue or action proposal need perform detailed analysis—others can simply indicate their neutrality.

Moreover, allowing review activities to include resolution discussion has significant advantages. In many cases, exploring resolutions provides useful additional insight into the nature of the issue and its interdependencies with other issues. Resolution discussion is also a "natural" part of review: as any attendee to a code inspection meeting knows, it takes conscious effort *not* to propose solutions to problems. Finally, resolution discussion during review can be more efficient: those people most qualified to review a resolution action are frequently part of the review, and incorporating resolution reduces or eliminates the additional meetings and scheduling typically required after a Fagan-style inspection.

For these reasons, we decided in the design of the asynchronous review environment to explicitly encourage the generation of resolutions to raised issues from participants, viewing this as an opportunity to exploit the power of group work that is unfortunately but necessarily lost from traditional synchronous meetings.

As a result of this research, we designed a data and process model for review that we have continued to use, with only minor changes, in all subsequent versions of CSRS to date. The process model consists of seven basic phases, as illustrated in Figure 4 and outlined in Section 2. The process model is coupled with a data model that describes the set of node and link types that can be defined and the legal relationships between them. A detailed description of this representation appears in (Johnson and Tjahjono, 1993).

3.2 CSRS I: Data-oriented instrumentation

From the start, we viewed measurement and instrumentation as a fundamental part of the design of CSRS. In the first version, we designed measurement from a "data-oriented" perspective: the measures were generated by querying the hypertext database at the conclusion of review for the number of nodes of a given type, or for those containing a specific value for a given field, or for those partaking in a specific relationship to other nodes.

Data-oriented measures are very useful: at a minimum, the number and severity of identified defects provides a first-order estimate of the quality of the software under review. Data-oriented measures can also reveal important characteristics of the review team and review process. For example, unproductive members of the review team might be identified (after a sufficient number of review instances) as those who contribute little, who contribute non-productively (by simply affirming comments made by others), or who use review for political purposes.

Finally, data-oriented measures can form the basis for controlled experimentation: given sufficient time and resources, an organization can fine-tune certain characteristics of review, such as the number of participants or the number of lines of code under review, by collecting data across a range of review instances and correlating these factors to, for example, the number of productive issues raised.

We experimented on the initial version of CSRS in the summer of 1992 by performing a review of a module called "Gtable" from the EGRET implementation. Gtable allows its clients to define a high-level abstraction for associating keys and values in a distributed environment, without concerning themselves with the details of replicating the tables to local hosts, maintaining consistency as updates are made by clients, and providing recovery procedures. The top-level of the Gtable implementation under review consisted of approximately 500 lines of Lisp macros and functions.

In the Gtable review, which lasted approximately three weeks, over 90 nodes were created by five reviewers during the private and public review phases. Following these phases, the moderator assessed the state of review to determine if a group meeting was necessary. It was found that a group meeting was required to resolve approximately only half of the 25 primary issues raised about the Gtable implementation.

This usage of the initial CSRS system convinced us that a computer-supported, asynchronous approach to review was legitimate and useful, and that the EGRET environment was an effective

platform for this application. This usage also revealed certain engineering issues that required immediate attention. During the fall of 1992, we devoted considerable effort to improving the user interface, the efficiency of both CSRS and EGRET, and the set of services provided (for example, facilities to automatically generate LaTeX hard-copy reports)

3.3 CSRS II: Elapsed-time instrumentation

A significant result from the first experiment was the insight that CSRS could be further instrumented to capture time-related data. By recording the time that a node is retrieved from the database by the client and the time that the node is closed, we expected to determine how long the participants spent reviewing each function, generating each issue, and so forth.

Capturing such time-related data could greatly enhance the analysis potential of CSRS. With this data, it would be possible to perform "fine-grained" analysis of review. For example, one could study the relationship between code size and time required for review at the grain-size of individual functions.

To explore this functionality, we enhanced CSRS with a counter for each node that accumulated the elapsed time that the node was retrieved. We then performed a review experiment on the "Nbuff" module mentioned above. The review involved 18 source code nodes, totalling 435 lines of code. Like the Gtable review, this experimental usage involved five participants over approximately three weeks. It generated 104 nodes that were eventually consolidated down to 19 actions. Six of these were controversial, requiring only a 35 minute meeting to resolve before beginning rework.

After review, we examined the timing data, and discovered to our surprise that it did not in all cases reflect the time spent in review. The essential problem is that recording the elapsed time that a node is retrieved by a client is only an indirect measure of the time spent reviewing.

Recall that CSRS is implemented using Unix and X windows, and consider, for example, the following scenarios:

- The participant takes a phone call while reviewing a node, or a colleague walks into the office and begins a lengthy discussion.
- The participant receives e-mail while reviewing a node, and switches to a simultaneously running mailer process to read and reply to it.
- The participant goes home for the weekend, leaving in the middle of reviewing a node with the intent to finish the review first thing on Monday. Since the participant has a dedicated workstation in his office, he leaves CSRS up and running.

In fact, the incorrect elapsed time values in Figure 5 for reviewer 1 on nodes gi*nbuff*read and gi*nbuff*make are due to the first and second scenarios.

These results led us back to the proverbial drawing board. We proposed and immediately discarded the idea of telling reviewers that they could not read their mail or
otherwise interrupt a CSRS review session; such a restriction would be impossible to enforce and the resulting data would be no less suspect.

Source code name	Size	Reviewer 1		Reviewer 2		Reviewer 3		Reviewer 4	
		Time	Iss	Time	Iss	Time	Iss	Time	Iss
gi*nbuff*read	25	2:28:47	3	0:49:08	2	0:28:39	0	0:40:43	1
gi*nbuff!pack-buffer	57	0:08:59	0	0:41:33	2		0	0:08:35	1
gi*nbuff!copy-and-pack	51	0:24:18	1	0:29:07	3		0	0:02:37	0
gi*nbuff!init-local-variables	19	0:09:06	1	0:16:57	1	0:02:49	0	0:24:00	0
gi*nbuff!unpack-field	46	0:10:19	1	0:32:28	0		0	0:08:22	0
gi*nbuff*make	20	2:04:44	5	0:36:18	2	0:14:50	0	0:13:38	1
gi*nbuff!make-link-label	34	0:01:39	0	0:40:45	1	0:00:22	0	0:04:15	0
gi*nbuff!unpack-buffer	26	1:46:29	3	0:03:42	0	0:04:22	0	0:17:22	2
gi*nbuff!make-field-label	18	0:00:42	0	0:25:39	0		0	0:03:06	0
gi*nbuff*write	33	0:06:50	1	0:09:54	2	0:07:48	0	0:14:38	0
gi*nbuff!delete-field-label	18	0:00:34	0	0:25:56	1	0:03:44	1	0:02:38	0
gi*nbuff!unpack-link	12	0:20:09	1	0:00:40	0	0:01:48	0	0:06:09	1
gi*nbuff*nbuff-p	12	0:05:11	0	0:01:08	0	0:04:16	0	0:04:15	0
gi*nbuff*node-ID	12	0:00:23	0	0:02:20	0	0:04:40	0	0:08:12	0
gi*nbuff!delete-link-label	31	0:02:08	1	0:02:08	0		0	0:02:38	0
gi*nbuff*read-hooks	5	0:00:11	0	0:07:53	1	0:00:32	0	0:00:35	0
position	12	0:00:50	0	0:00:12	0	0:03:12	1		
gi*nbuff!node-ID	4	0:00:20	0	0:00:08	0	0:00:49	1	0:00:13	0
Total	**435**	**7:51:39**	**17**	**5:25:56**	**15**	**1:17:51**	**3**	**2:41:56**	**6**

Figure 5. Elapsed time data generated from the Nbuff review.

3.4 CSRS III: Improved elapsed-time instrumentation

In the next version of CSRS, we redesigned the time-based instrumentation to address this issue. Instead of simply maintaining a counter with each node that would accumulate a single elapsed time value, we implemented a more sophisticated and general purpose event-based timestamp facility within EGRET. This facility allows EGRET-based applications such as CSRS to insert timestamp calls at strategic points within their code to record the occurrence of arbitrary events of interest. (For example, when the user opens a node, writes a node, traverses a link, and so forth.) EGRET provides the underlying mechanisms to fast-cache the sequence of timestamped events at the local client during the session, and write the cache out to the server database at disconnect time. Timestamping inevitably incurs some overhead, and it is possible that over-zealous insertion of timestamp calls by an application can visibly degrade the responsiveness of the system. (We have not, however, observed degradation in performance due to timestamping in CSRS or two other applications developed using EGRET.)

Using this redesigned instrumentation, we performed a review experiment on a prototype newsreader system called URN. The review involved 53 source code nodes, totalling 478 lines of code. This review lasted 10 days, generated 75 reviewer-based nodes, and 35 issues. We timestamped database connection and disconnection, node
reading, writing, creation, deletion, and setting the status field in source code nodes (the status field is used by reviewers to explicitly signal when finished reviewing a node).

Figure 6 illustrates a portion of the time-stamped event log for the URN review experiment. Participants did not report any noticable degradation in the responsiveness of the system from the timestamping mechanism.

ID	Operation	Node	Name	Date	Time	Event Interval	Screen	Misc
60	connect		csrs	20-May-93	10:09:48			Private
60	summarize-sources		Summary-sources	20-May-93	10:10:13	0:00:25	Summary	
60	read-node	350	uin*key!close-article	20-May-93	10:10:20	0:00:07	Source	
60	close-node	350	uin*key!close-article	20-May-93	10:30:49	0:20:29	Source	
60	close-node	362	overview	20-May-93	10:30:50	0:00:01		
60	disconnect		csrs	20-May-93	10:30:50	0:00:00		Private
60	summarize-sources		Summary-sources	20-May-93	10:36:21		Summary	
60	connect		csrs	20-May-93	10:36:21	0:00:00		Private
60	read-node	350	uin*key!close-article	20-May-93	10:36:30	0:00:09	Source	
60	close-node	350	uin*key!close-article	20-May-93	10:38:59	0:02:29	Source	
60	summarize-sources		Summary-sources	20-May-93	10:39:02	0:00:03	Summary	
60	read-node	260	uts*article*make	20-May-93	10:39:19	0:00:17	Source	
60	summarize-sources		Summary-sources	20-May-93	10:39:39	0:00:20	Summary	

Figure 6. A portion of the event log generated during the URN review. Over 1000 events were logged during the URN review.

The timestamp mechanism significantly improves the CSRS instrumentation. First, timestamps can recreate the elapsed time data we obtained through the previous mechanism.

More importantly, timestamps allow us to assess the accuracy of the elapsed-time information, by helping detect the occurrence of review interruptions due to scenarios like those noted above. This is accomplished by calculating the inter-event intervals. The reasoning goes as follows. If CSRS is instrumented "correctly", then, under typical usage patterns, events should be generated relatively frequently and consistently. A significant interruption in review, due to answering the phone, leaving the office, and so forth can be detected by an abnormally large value for an inter-event interval. However, if CSRS is instrumented "incorrectly" (i.e. the timestamp calls are too sparsely distributed in the application code), then not enough timestamps will be generated to distinguish such interruptions from normal patterns of timestamp generation during review.

Figure 7 shows a histogram of inter-event intervals collected during one phase of the URN review. This data shows that timestamps were generated less than 30 seconds apart over 80% of the time, less than a minute apart over 90% of the time, and less than three minutes apart over 98% of the time.

It is important to be precise about what can and cannot be inferred from inter-event interval data. While a sequence of small inter-event interval values (say, less than or equal to 10 seconds) does effectively indicate essentially uninterrupted use of CSRS, the converse is not true: a high inter-event interval value does not necessarily indicate the occurrence of interruption.

It is possible, for example, that a reviewer could simply stare at a section of source code for many minutes without performing any CSRS-related action that would trigger an event, although this seems somewhat improbable. A more likely scenario is one in which a reviewer spends a lengthy period of time carefully composing an issue in a CSRS editor buffer without retrieving, saving, traversing, or otherwise interacting with CSRS.

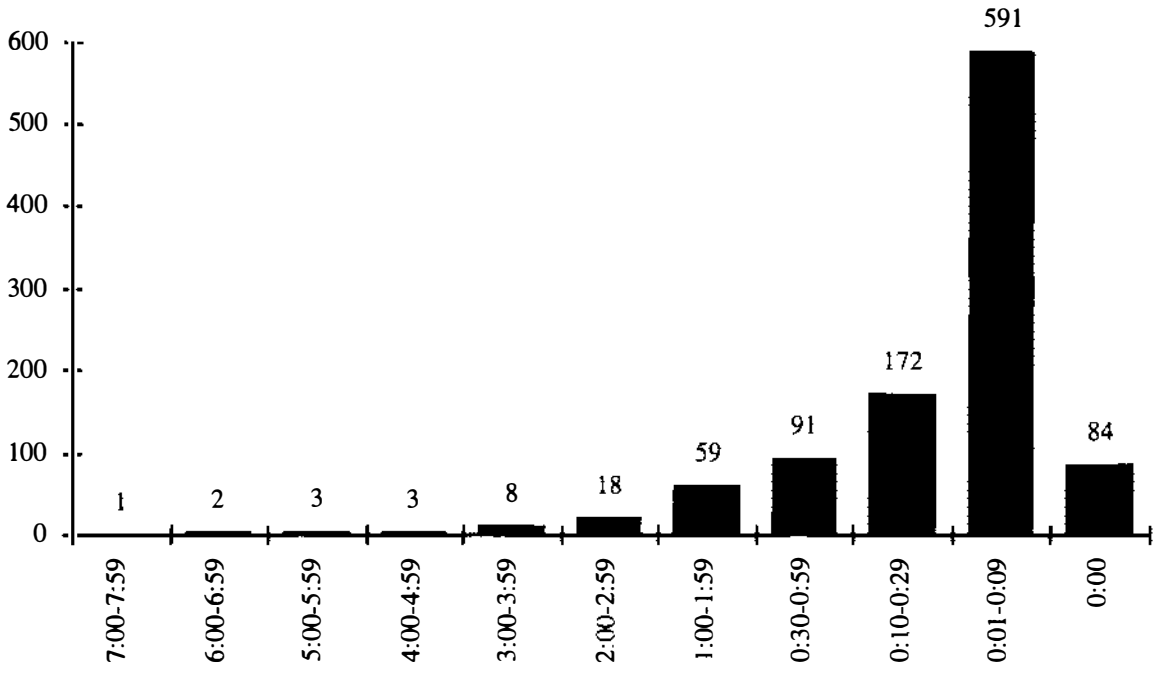


Figure 7. A histogram illustrating the frequency of inter-event intervals in the URN private review phase. For presentation purposes, a single interval with value 20:29 has been omitted from this figure.

With the timestamp calls used in this experiment, the event interval data would be identical to an alternative scenario in which the reviewer spends a couple of minutes composing the issue, leaves for lunch, and then returns and saves the issue to the database.

While the event interval data might be identical in these two cases, one would hope that the two issues would differ qualitatively. Thus, an appropriate way of regarding event interval data is as a means to pinpoint areas of ambiguity in the timestamp data: places where further study is needed to assess the accuracy of the information.

As a concrete example, in the URN review, the highest recorded interval was one of 20 minutes and 29 seconds. Figure 6 illustrates a portion of the event interval log containing this value, which occurred between 10:10am and 10:30am on May 20, 1993. Notice the context in which this event interval occurred: the reviewer connected to the system at 10:09 a.m., retrieved the node containing the source code uin*key!close-article at 10:10 a.m., and then closed the node and immediately disconnected at 10:30 a.m. Given that this particular function is only five lines long and reasonably straightforward, and that no other activity occurred during this entire session except to retrieve and close this node, the interval value is suspect. An interview with the reviewer confirmed that this particular elapsed-time value for uin*key!close-article could be thrown out, since the reviewer was indeed interrupted during this time period.

In a very recent version of the system, we incorporated a timer-based mechanism that wakes up once-per-minute and checks to see if any low-level editor activity (such as a keystroke or mouse click) has occurred in CSRS during the preceeding two minutes. If such activity occurred, it writes out a generic "busy" timestamp event. This mechanism may have substantially solved the

problem of detecting idle time, since we believe it to be extremely unlikely that a reviewer will simply stare at a CSRS screen for over two minutes without so much as scrolling a window.

3.5 CSRS III: Process-oriented instrumentation

Moving to an event log-based instrumentation mechanism has an even more profound impact than improving the accuracy of the elapsed-time data: it instruments the *process* of review at a fine-grained level. By analysis of the event log, it is possible to reconstruct the sequence by which a reviewer traversed the hypertext database, even reconstructing the set of nodes displayed concurrently in different windows at any point in time.

Such forms of analysis carry with them new and significant quality assessment issues. For example, to reconstruct this process-level information accurately, users much not change the default screen orientation manually during review.

However, we believe this process-level analysis of CSRS review has significant potential to aid in process maturation of FTR, as developed in the SEI Capability Maturity Model (Humphrey, 1989). By analyzing the sequence of actions taken by reviewers as they perform FTR, we expect to improve the user interface to CSRS and the set of services provided. More fundamentally, we believe that high-quality, fine-grained, process-oriented measures, when combined with high-quality, fine-grained data-oriented measures, will provide new and efficient support to organizations in designing and improving FTR methods custom-suited to their organizational and application-level needs. This will bring our research full-circle, as we use the instrumentation data to return to the initial focus of our design: a data and process model more efficiently suited to on-line, collaborative review.

4. Current status

We are currently using CSRS in-house on a nearly continuous basis, not simply to further refine the paradigm, but because we are firmly convinced that it provides the most cost-effective way of improving the software quality of CSRS itself and the other applications under development in our research group.

While we have not yet collected enough quantitative data under controlled conditions to present statistically significant conclusions concerning the process and products of FTR using CSRS, our data collected to date does support some general observations.

First, the data indicates that CSRS review appears to proceed at approximately the same rate (100-250 lines of code/person-hour) as that reported for Fagan-style code inspection. However, as alluded to above, CSRS review captures the review "discussion" much more completely than can a scribe manually writing notes during a meeting, and also includes discussion of resolutions that are typically excluded from manual review methods. Thus, more information in greater detail is produced in the same amount of time using CSRS.

To provide some perspective of the range of artifacts captured during review, we manually categorized the types of issues and commentatary raised during the URN review, and found that approximately 20% were either: design rationale related; clarified the specifications or behavior of the application under development; or clarified the specifications or behavior of EGRET or some

other underlying infrastructure (such as the source language). The presence of such captured artifacts (which would not typically be recorded in a conventional review–indeed, they would be viewed as a "digression") have helped us to improve the specifications and documentation of EGRET and other systems, and provided significant aid to the developers.

The current users of CSRS completed a questionnaire asking subjective questions about their satisfaction with the system. Aside from the generic eternal user plea for faster response times, high satisfaction was indicated. (These responses, however, are from a small and biased population.) Better subjective data will soon be forthcoming, however, as CSRS is scheduled for use in an external software development group during the summer of 1993.

In conclusion, to summarize our lessons learned:

- CSRS formal technical review requires a different data and process model than those designed for manual FTR methods.
- User satisfaction with CSRS appears high.
- CSRS does not appear to appreciably change the rate of review as compared to traditional FTR. However, this rate comparison is misleading, since CSRS review encourages resolution activities that typically take place downstream from traditional FTR.
- CSRS review captures a much broader range of information than application-level defects, the focus of traditional FTR.
- The EGRET timestamp mechanism used in CSRS provides both fine-grained data on the process and products of review, and supports quality assurance activities on elapsed time data. However, this quality assurance currently requires manual post-processing, analysis, and interpretation.

5. Acknowledgments

Support for this research was provided in part by the National Science Foundation Research Initiation Award CCR-9110861 and the University of Hawaii Research Council Seed Money Award R-91-867-F-728-B-270. We thank the anonymous reviewers for many helpful comments that improved the quality of this paper.

6. References

L. Brothers, V. Sembugamoorthy, and M. Muller (1990): ICICLE: Groupware for code inspection. In *Proceedings of the Conference on Computer-Supported Cooperative Work 1990,* pp. 169-181. ACM Press.

Lionel E. Deimel (1990): *Scenes of Software Inspections: Video Dramatizations for the Classroom.* Software Engineering Institute, Carnegie Mellon University.

Robert Dunn (1990): *Software Quality: Concepts and Plans.* Prentice Hall.

Michael E. Fagan (1976): Design and code inspections to reduce errors in program development. *IBM System Journal,* 15(3):182--211.

Michael E. Fagan (1986): Advances in software inspections. *IEEE Transactions on Software Engineering,* SE-12(7), pp. 744-751.

D. P. Freedman and G. M. Weinberg (1990): Handbook of Walkthroughs, Inspections and Technical Reviews. Little, Brown.

John Gintell, John Arnold, Michael Houde, Jacek Kruszelnicki, Roland McKenney, and Gerard Memmi (1993): Scrutiny: A Collaborative Inspection and Review System. In *Fourth European Software Engineering Conference*, Garwisch-Partenkirchen, Germany, September 1993.

Watts S. Humphrey (1989): *Managing the Software Process.* Addison-Wesley.

Philip M. Johnson (1992): Supporting exploratory CSCW with the EGRET framework. In *Proceedings of the Conference on Computer-Supported Cooperative Work 1992,* ACM Press.

Philip M. Johnson and Danu Tjahjono (1993): Improving Software Quality through Computer Supported Collaborative Review. In the *Third European Conference on Computer Supported Cooperative Work,* Milan, Italy, September, 1993.

Roger S. Pressman (1992): *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Inc.

Glen W. Russel (1991): Experience with inspection in ultralarge-scale developments. IEEE Software, (9)1.

G. Gordon Schulmeyer and James I. McManus (1987): Handbook of Software Quality Assurance. Van Nostrand Reinhold.

U. Wiil and K. Osterbye (1990): Experiences with hyperbase-a multi-user back-end for hypertext applications with emphasis on collaboration support. *Technical Report 90-38,* Department of Mathematics and Computer Science, University of Aalborg, Denmark.

Edward Yourdon (1989): *Structured Walkthrough.* Prentice-Hall, Fourth Edition.

Quality guided Programming: Integrating code reviews with metrics analysis of code

STEFAN BIFFL THOMAS GRECHENIG

Department of Software Engineering,
Technical University of Vienna,
Resselgasse 3/2/188, Vienna,
A-1040 Austria, Europe

EMail: Biffl@eimoni.tuwien.ac.at
Tel.: ++43-1-58801-4082
Fax: ++43-1-504 15 80

Abstract

Quantitative evaluation of software systems has not yet been accepted by practitioners. Early expectations especially into code analysis have not been met so far. Among several reasons for the rare use in practice we suppose a lack of empirical data, a dominant focus in research on formal aspects, an unreasonable embedding in the development process, and a lack of flexibility and usability of code measuring tools. The following paper deals with the outline of a process model for quality assurance during the coding phase providing human reviews as well as quantitative evaluation. The model is based on the idea of permanently adapting measuring guidelines to the goals of a project what will result in a metric and review guided coding cycle.

We postulate that quantitative evaluation can work in practice if metrics, project constraints and management goals are matched within a local process of collecting empirical data. Experiences of a student team workshop are reported. The overall results are: in the cases, in which the process model and the code metrics had been applied, it was possible to achieve positive trends towards fulfilling given goals. The application of code metrics can help to focus the precious resources of quality assurance (QA) personnel on probably inferior pieces of software. Before the proposed process model can be useful within an industrial environment, basic QA standards must be established within this environment.

Keywords: quality assurance, process model, software metrics, quantitative evaluation of software

The authors

Stefan Biffl and Thomas Grechenig are assistant professors at the department of software engineering at the Technical University of Vienna, Austria. They both received MS degrees in computer science and Ph.D. degrees in software engineering. Stefan Biffl teaches quality assurance and has a special interest in code metrics. Thomas Grechenig is doing research in requirements engineering and user interface design. They are both involved in organizing a software engineering workshop for 500 students. They have experiences as consultants in industrial (research) projects concerning quality assurance and reuse of software.

1. Introduction

Software Quality Assurance (QA) has been a main issue with developers in the industry for at least ten years now. Paradoxically the actual quality assurance groups within project teams have seldom had the chance to prove their worth; there have been almost no cases where they were allowed to play their part in helping project management. The typical QA group consists of highly trained individuals with a strong bias to the theoretical side of things. In the average project environment this rather hinders than helps cooperation; QA people often find that their skills are only partly used. A discipline of QA that is at least sometimes lived is the methodical review of software documents. In the industry the least-used methods within QA are certainly quantitative measures to judge the quality of code: these software metrics, often awkward to use and yielding results that are far from the vital project information needed by project managers, never found widespread use.

Meanwhile research has been done to overcome the gap between the theory of software metrics and their application. [Basi87], [Paul92] suggested process models for implementing quantitative quality assurance. Some case studies on practical experiences have been reported (e.g. [Kitc89], [Hon 90], [Sama91]). Nevertheless mainstream research still behaves as if it were necessary to market the use of software metrics as such ([Sieg92]). This is a clear sign that despite the work mentioned above, metrics and especially code metrics have still not yet proved to be successful in practice. Among the various reasons we find:

A focus on formal aspects: Many collections of formulas have been suggested (e.g. [Prat84], [Adam87], or [Cote88]) that should map software characteristics like complexity, information contents, structure of data flow, structure of control flow. Obviously most of the metrics that have been suggested during the last 15 years are of a certain mathematical and statistical beauty. But they did not really bridge the gap to the needs of software engineers at work. Practitioners want project oriented information. Mostly this is much more than aggregating measures on single code lines.

A lack of empirical data: Though many experiences have been reported, no general theory has been derived from them. Probably this is impossible within the variety of today's development conditions. Results are reliable only within a certain environment and cannot be transferred easily. Up to now parametrized theories based on empirical results have not been derived, that would allow for a reasonable customization and application in another development environment.

A misplaced embedding in the development process: In practice code metrics as means of quality assurance are applied usually before a piece of code is on the real job but after fulfilling functional requirements and testing. This is why many developers feel about code metrics as if they were some unnecessary appendix. Quantitative requirements to code have to be embedded in the development process together with other quality assuring activities like e.g. a reviewing procedure. This process serves the purpose of quality guided development which can establish a sort of local standard ([Grec92]).

A "masses of syntactic data reveals semantic information" assumption: Though often subconsciously expected, it is practically impossible to gather information which is above the level of static semantics, e.g. on the quality of mnemonics or the readability of code comments, relying on code metrics only. Code metrics are no total substitute for other quality assurance activities like reviews by colleagues.

A lack of flexible tools: Metrics are most effectively defined, adapted or modified by each quality assurance group on its own by collecting *local* information and extracting *local* standards. One condition is a certain comfort in defining metrics and actual measuring. In fact this uncovers another important but underestimated technical reason for practitioners' low acceptance: a lack of flexibility and usability of the available tools.

Part 2 deals with a quality assurance process designed for adapting code metrics to local conditions. An application example dealing with maintenance efforts is described in part 3. In part 4 extended experiences from student projects as well as a selected example from an industrial development environment are reported. An enhanced version of the process is derived integrating code reviewing techniques: code reviewing is the general activity, quantitative evaluation is regarded as means of automatic reviewing for a certain class of review criteria..

2. A process for creating, assuring and standardizing code quality through software metrics

Quantitative code analysis can be much more successful in practice if

- it is applied together with other quality assurance activities,
- it is properly embedded in the process of quality-guided development with respect to technical and organizational aspects,
- it focuses on gathering information and assuring standards which are *local* with respect to a certain domain of projects within a developer's business environment.

In the following we sketch a practical model for both quantitative and qualitative analysis which defines a process of quality-guided programming.

Adjusting metric definitions to project goals and standards is an iterative process; it means collecting experience by measuring actual code and slowly minimizing the gap between the abstract project goals on one hand and the measuring results (metric values) on the other. It is a process of matching metric values and project constraints which are to be guided and performed by the quality assurance group. More generally spoken, this cycle process serves as a mediator between the project management and its programmers.

Metrics will mainly be useful if applied locally with respect to determined types of projects and organizational and economic premises. To build up and maintain significant information requires a person, who is close enough to the software development team to be sensitive about their needs on one hand, but who is on the other hand not involved so much as to get overrun by deadline pressures like it often happens to the programmers.

We call this position metric-engineering. The metric-engineer is a person within a quality assurance team (see fig. 1). He/she supports the project manager in planning quality issues and measuring the actual software which programmers produce. He/she holds the same organizational position as the person who is responsible for the process and quality of code reviewing. Collecting information through metrics needs an experienced software-engineer with supplemental knowledge of formal methods of quality assurance and a solid background in literature of software metrics.

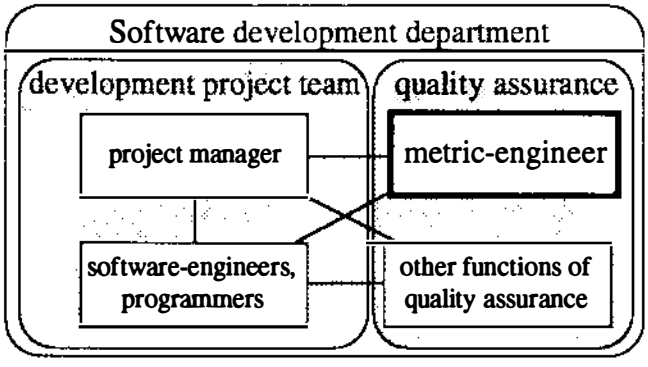


Figure 1: The metric-engineer's organizational embedding in a software development department

Figure 2 shows a code quality assuring process in which the metric engineer holds an important position. Three roles have to be considered: The project-manager (PM) serves as an expert for the goals of the whole project, a metric-engineer (ME) knows about metrics and models for rating the possible alternatives to achieve the project goal and the software-engineers (SE) implement the product by following the technical model and a set of given metrics.

a) Analysis of goals: The process starts with the PM's analysis of the project goals (e.g. reusable code, high quality documentation, meeting deadlines, etc.), identifying the range of possibilities to achieve these goals and setting levels of necessary quality requirements for these alternatives.

b) Goals/questions/metrics (GQM, see [Basi87]): The PM and the ME meet to deduce from the goals questions, criteria and quantifiable categories, which can yield quantitative information about the achievement of the goal. From this catalogue of questions a set of metrics will arise, which has to be

measured in order to answer the questions. On the basis of these metrics the alternatives to meet the questions are evaluated, cost and benefits are estimated.

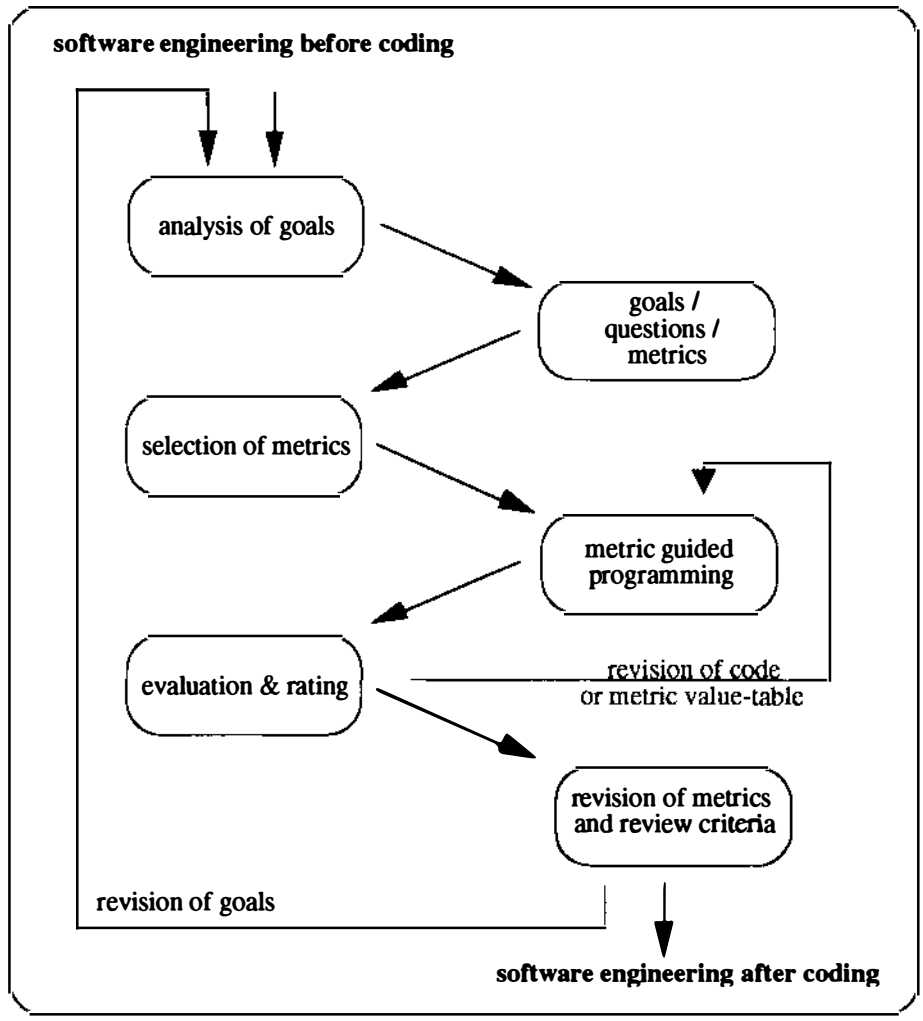


Figure 2: A process for gathering, adapting and controlling standards for metric guided coding

c) Selection of metrics: The PM and ME decide, which alternative to use as their first choice. For each metric identified in the previous step, they establish value-tables, which should be met when the task is finished: minimum, maximum, and a favorite value range. The metrics get weighed, such as to make the relative importance meeting their values obvious. At the end of this step appropriate measuring-tools are either selected or newly built..

d) Metric guided programming: Software engineers and programmers are informed about their subtasks and the metric suite defined before. Usually the metric engineer provides them with the generated measuring tool. The software engineers start planning, designing, coding, and testing. They should know the principal goals of the project (e.g. low maintenance costs and fast reaction to bugs identified at the time of production).

e) Evaluation and rating: While the SE works on the solution of his problem, the ME measures or helps to measure the newly developed pieces of code and provides feedback to the SE such as to enable him to adapt his style of working to the metric-tables if necessary. Partly this should already happen in the phase before through the programmers' self control. During the phase at hand metric-tables can get changed by the ME too, in order to fit "coding reality". In both cases the loop back to phase "metric guided programming" is taken. When the SE's results and the metric-model meet, the cycle advances to the following step.

The benefit in comparison with the usual review process is: With the help of measuring results the ME can concentrate on the parts of the product which seem to need most attention by QA. In order to be able to fine-tune his measuring tools for the various projects in the software development department, the ME will certainly need an environment which enables him to shape his metrics in a simple way.

f) Revision of metrics and review criteria: The degree of fitness of the PM's qualitative requirements is compared to the actual results of maintaining the quantitative indicators. The border-values and weights of the metrics get adapted. These new metric-tables can later be used in step two of a similar following project, where, if project constraints are comparable, the previous metric-table can serve as a useful initialization (see feed-back loop in fig. 2).

Within the whole process of quality-guided development one can regard the so-called quantitative evaluation as one special form of review. To say it in other words: anything that can be automatically measured by software metrics could be a result of human review, too. But reviewing is costly and time consuming. So one of the goals of any QA group should be to move criteria control as much as possible from human reviewing to metric analysis (see fig. 4).

Putting the above cycle into actual work requires a tool which can be adapted easily by the metric engineer. It is obvious that adoption is somehow permanent, therefore the ME and not only the tool vendor has to be capable of generating new tools. The best place for a tool of that kind would be that of a component in some development framework (e.g. a CASE-tool).

3. The code quality assuring process: an everyday example

The previous part of this paper described the general outline of a metric guided coding process. In the following a concrete and simple example will be sketched. The scenario at hand is a contract of a software developer whose client is conscious about possible maintenance costs. The process described above can be realized by the following six steps:

a) Analysis of goals

Assume that the PM has to plan a specific development project. In the contract it says that in case of high maintenance costs they have to be covered by the supplier. Therefore the major requirement is low maintenance costs over the product's life-span. Another requirement is a guaranteed repair-time of not more than one week in case of errors. The PM can define these requirements as more or less informal qualities. His/her message to the quality assurance group will be: Software must be cheap in maintenance, and short in errors that cannot be eliminated quickly.

b) Goals/questions/metrics

With a first glance at the PM's goals the ME can see that they are too informal to be tested. They are also too imprecise to be used as guidelines for the production process. After some arguing on the issues of maintainability and "quick elimination of errors that occur" they agree on the criteria: 1) Any procedure has to be well documented. 2) Complex parts have to be documented intensely. They decide for lines of comment as a measure of documentation. To identify "complex parts of code" they take the measures of total complexity according to [McCa76] with respect to one statement .

c) Selection of metrics

The ME and PM set up a metric-table to specify their ideas of step b) (see table 1):

Measure	mini mum	maxi mum	favorite value range	weight
comments at each procedure head	1	40	5 - 8	0.3
comment lines at each statement which enclose code whose (McCabe) complexity exceeds 4	1	10	3 - 5	0.4
ratio comment lines to total lines	30%	60%	40% - 50%	0.3

Table 1: Sample metric-tables as a constraint for coding activities targeting at lower maintenance costs

d) Metric guided programming

The metric engineer hands a measuring tool allowing the quantification of code due to the criteria developed in c) on to programmers. That tool prints out warnings, if the standards defined in the metric-table are not met.

e) Evaluation and rating

Either in regular intervals or whenever the programmers deliver a piece of code the ME measures the product, interprets the numbers, and discusses his results with the SE. This may lead to the SE refining his code (back to previous phase) or to the acceptance of the part of the product which is currently discussed by the ME (advance to next step).

f) Revision of metrics

After completion of the project the PM and the ME meet again to discuss the success of their policy. They recognize that the code of the recent project is actually better documented than it is usually the case in the department. Due to the selective reviews of the ME during the project some major sources of errors as well as of confusion were removed through documentation or redesign. On the basic questions of maintenance cost and time to fix errors they agree on having a log-book as documentation of these items on one hand and on meeting for the purpose of a review every quarter on the other. The experience gained in this process will be used to raise the quality of tuned metric-tables and thus forming a department-wide learning curve.

4. Application and integrating quantitative evaluation as a special means of reviewing

At the Vienna University of Technology a software engineering course is organized for 500 students. During one academic year software systems are developed in teams of 5-8 people from the phase of requirement engineering to the phase of preparation for maintenance. The period of coding takes two months within that time span. The development platform has been object oriented (OO) Pascal including libraries for an OO (relational) database access and OO-user interfaces. The typical student project consists of 10000 to 15000 lines of code.

Senior students are working as tutors. They advise and grade the students. Each tutor has to guide between two and three teams. Four tutors (out of 24) and eight teams have been selected to take part in an experiment with metric-guided programming as was explained in figure 2. The roles of the process were defined as follows: assistant professors took the part of the goal-setting project manager, the tutors acted as metric engineers who measured the programming results, i.e. work of the students who held the part of the software engineers.

4.1. Applying the process model in a university development environment

a) Analysis of goals

The goal for the student teams was to get a neat object oriented design with respect to strict information hiding, adequate inheritance, polymorphism, and possible reuse.

b) Goals/questions/metrics

This is the difficult task in metric engineering: finding reasonable questions and metrics which yield data being in close connection with the actual goals. Assistant professors and tutors agreed on four *questions* and translated them into a set of simple *metrics* which were easy to measure (see table 2).

The four above *goals* have been translated into the following four *questions*:

1) *Information hiding:* How many accesses to an attribute come from outside the defining object (which is a violation of information hiding possible in any hybrid OO language)?
2) *Inheritance:* Are objects usually defined by inheritance or are they created from scrap?
3) *Polymorphism:* Are objects coded in a way that descendants can change the behavior of methods of ancestors without completely rewriting these methods (late binding)?
4) *Reuse:* Are objects (designed to be) used in more than one project?

Since it is clear that the above questions cannot be answered by static code metrics only, we have chosen a pragmatic approach. We proposed *metrics* that should help identifying those parts of the code which were most likely to violate the *goals*:

1) Check for any accesses of attributes from the outside the object and let them effect the overall score very negatively (see table 2). Attributes should be accessed at best by only one or two methods which act as an interface to the attribute.

2) Count the depth of the inheritance tree for each object. Objects without ancestor are not counted.

3) Polymorphism is not easy to check. We tried to give simple hints: Polymorphism needs good information hiding, and if done properly leads to a few lines of code for newly defined methods. But this evidence is weak and is to be rated low (see table 2).

4) Interproject reuse can't be measured in a short term. Again only vague criteria can be applied: Reusable objects' methods should be short (in lines of code), and should have an appropriate interface (not too many parameters).

c) Selection of metrics

In table 2 you can find the selected *metrics* including the values which emerged from our early estimations after measuring and discussing with students and tutors. The values changed within the first two cycles of the measuring process and the recalibrating process, but stabilized then.

Measure	mini mum	maxi mum	favorite value range	weight
number of methods which access a specific attribute (question 1, 3)	1	10	2-3	0.4
depth of an object's inheritance tree (question 2)	2	12	4-8	0.2
length of methods in lines of code (question 3, 4)	2	30	5-15	0.3
number of parameters in method (question 4)	0	7	2-5	0.1
number of accesses to an attribute from outside of an object (question 1, 3)	0	99	1-99	- 1

Table 2: Sample metric-tables as a constraint for coding activities which target a certain OO-structure of the system (min-, max-, fav-value ranges define a trapezium-shaped function mapping the actual value to [0,1], see [Berr85])

d) Metric guided programming

A measuring tool evaluating code pieces provided the students with an immediate feedback to their developing results. The input for the tool were files of source code, the output was a report file containing a listing of the objects in the source sorted for each file by the cumulative metric score of the object (average of the scores for its attributes and methods according to the measures in table 2).

e) Evaluation and rating

Every other week when students had to hand over pieces of the final product they were reviewed by their tutor. These review results as well as the metric values were collected and compared with those of the other teams as well as also with the control group. Works that scored outstandingly (positively or negatively) were examined thoroughly to get an idea how measured values were connected with "real quality" of the code substance.

In the beginning we found some "awful" pieces of software (due to the experienced tutor review) scoring very well (in metric´s evaluation). These pieces came from programmers who had shaped their code towards optimizing the selected metrics. Our reaction made clear that any extreme piece of code no matter how good the metric score was would be examined closely and very bad code (according to human review results) would be rejected.

g) Revision of metrics and review criteria:

After the end of the workshop we measured and reviewed the results of other student teams which were said to be very good by their tutors. Their code metric score usually was average (see fig. 3). Most of the teams who coded according to the measures had average or above average review ratings and some of the best works came from them.

In fig. 3 an overview of the grading of 29 programs is shown. The meeting of the goals and questions from process steps a) and b) has been tested (true quality according to the review grading of experienced programmers) versus the

scoring on the selected metrics table (see table 2). In our metric guided programming experiment nine teams took part while 20 teams developed their code without specific instructions. Obvious theses of the evaluation were:

1) Most programs scored average regardless of the true structural quality of the code. Programs with remarkably bad scores got them due to the negative score for violation of the information hiding principle.

2) The metric ratings for the informed teams in the experiment usually were at least as good as the corresponding rating by the tutors.

3) The metric ratings for the informed teams in the experiment usually were higher than the rating of control teams' programs of comparable quality judged by quality assurers (see fig. 3).

4) Programs with a bad rating often turned out to be bad and never to be very good.

5) Very good programs usually got average metric ratings.

The experiment strengthens the common hypotheses that it is not possible to distinguish between very good and average code by means of code metrics only, no matter how cleverly they may ever be picked. However, it is possible to identify "candidates" for very bad parts of the software.

When we interviewed students on the mandatory use of code metrics in software development the programmers' reaction was ambivalent. In a way they expressed reluctance against being too strongly controlled by "unreasonable" rules. Obviously they have to be involved in the metric design procedure, too, in order to make them understand code measuring not only as a hinderance their creativity.

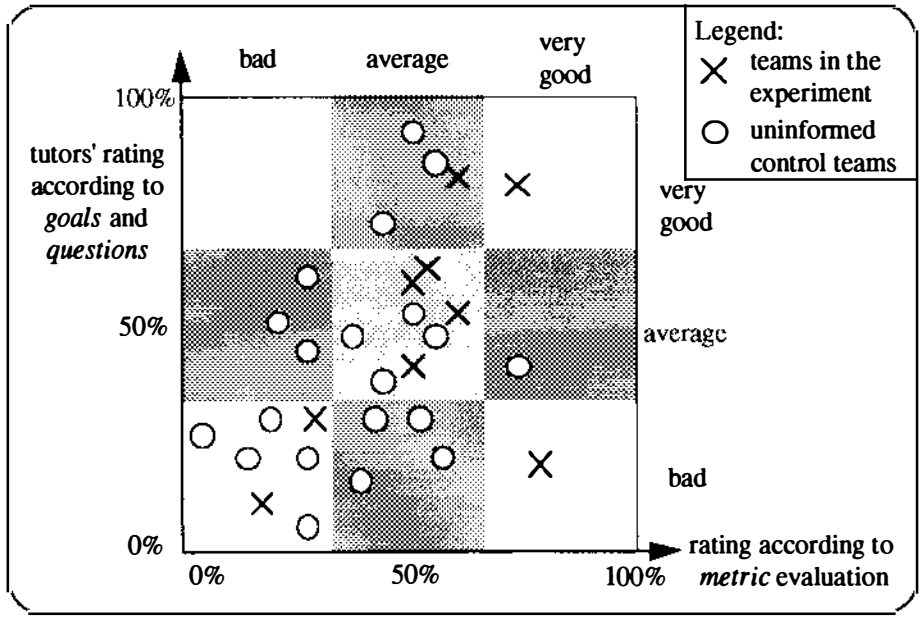


Fig. 3: Results of experts review evaluation versus automatic metric evaluation

4.2. An attempt of application in an industrial pilot project

During a project dealing with a hardware-system change from a traditional host environment to a modern client-server architecture we consulted the development of the future software engineering concepts and strategies of the client (banking). Within a pilot development project we applied our model:

The project manager planned to promote cost-effectiveness by reuse of object oriented software implemented in C++. The project group was trained on metric basics, the metric guided programming process, and the use of the measuring device.

The experiences correspond to what has been discussed in literature in other attempts for reasonable QA in practice. The first experiment proceeded at code which was written for training, benchmarking and testing reasons. There the new criteria were easily accepted.
As soon as the first milestone appeared in a "non-experimental" project time pressure turned up and developers did what they were used to do: They forgot software engineering and concentrated their power to make their code run. The very first development strategies to be neglected were quality reviewing, documenting the latest changes, as well as following the proposed strict configuration management procedure. The automatic measurements were performed but the results did not lead to reusable code. It is obvious that the practical use of quantitative code evaluation depends on the interest and decisions of project management at the moment of a milestone deadline/quality assurance conflict.

The industrial application of code metrics or metric guided programming is surely confined to quality intensive projects where proper quality assurance has been institutionalized and lived for some time. Within these constraints a well organized reviewing procedure dealing with chosen code samples and using quantitative measures (reviewing it all and reviewing after any change is too time consuming for most commercial projects) will bring the process of metric guided programming to its best degree of effectiveness (see fig. 4).

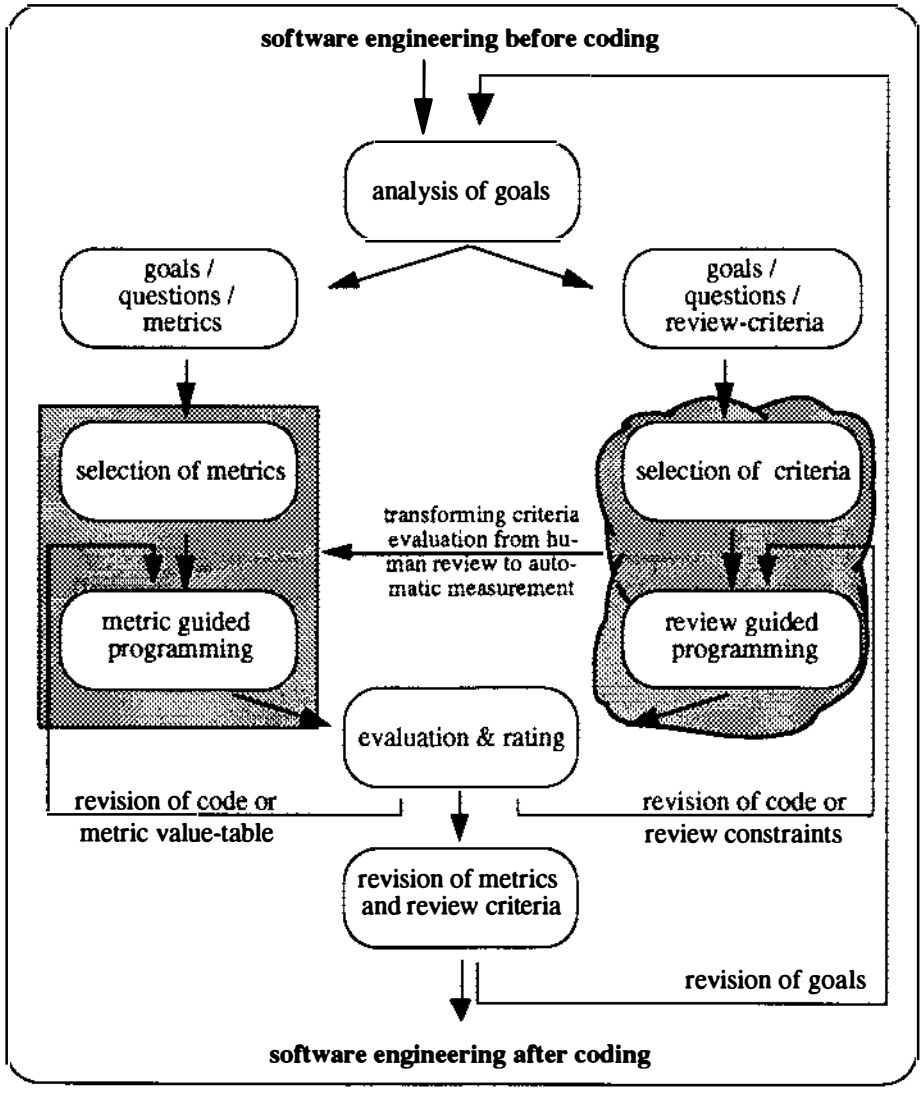


Fig. 4: Integrating quantitative evaluation as special means of reviewing

5. Conclusion

Within this paper we outlined an approach for making quantitative evaluation of software systems. This approach is based on the idea of developing coding rules locally. A process model has been described which assures quality-guided programming by using the techniques of quantitative evaluation. Its implementation affords the role of a metric-engineer, and a reasonable integration of quantitative code evaluation within a process of qualitative evaluation (reviewing) as well as a flexible tool for measurement.

The process model described in this paper is a methodological contribution towards the practical expediency of software metrics. Matters of organization and psychology are the subsequent "real" problems of the domain. More than the method or process itself an adequate educational preparation of the involved developers is necessary.

Bibliography

[Adam87] Literature Review on SW-Metrics, Adamov R., Baumann P., Institut für Informatik der Universität Zürich, Okt.1987

[Albr83] SW-Function, Source Lines of Code, and Development Effort Prediction: A SW-Science Validation, Albrecht A. J., Gaffney J. E., IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, pp. 639-648., Nov 1983

[Basi87] Tayloring the SW process to Project Goals and Environments, Basili V., In Proc. of the 9th Int. Conf. on SE, ACM, 1987

[Berr85] A Style Analysis of C Programs, Berry, R. E., Meekings B.A.E., Communications of the ACM, Vol. 28, No. 1, pp. 80-88., Jan 1985

[Bind90] Field Experiments With Local Software Quality Metrics, Binder L.H., Poore J.H., Software - Practice and Experience, Vol. 20(7), p.631-647, 1990

[Boeh84] Software engineering economics, Boehm B.W. IEEE TSE, Vol.SE-10,No.1,p.4-21 Jan. 1984

[Boeh88] Understanding and controlling software costs, Boehm B.W.IEEE Transactions on Software Engineering, Vol.14, No.10, p.1462-77, Oct.1988

[Bria92] Providing an Empirical Basis for Optimizing the Verification and Testing Phases of SW Development, Briand L.C., Basili V.R., Hetmanski C.J., Proc. on Int. Symp. on SW Reliability Engineering, North Carolina, USA, Oct 1992

[Cont86] SW Engineering Metrics And Models, Conte, Dunsmore, Shen, Benjamin/Cummings, 1986

[Cote88] Software metrics: an overview of recent results, Cote V., Bourque P., Oligny S., Rivard N., J. Syst. Softw., Vol.8, No.2, March 1988, p.121-31.

[Gill91] Cyclomatic Complexity Density and Software Maintenance Productivity, Gill G.K., Kemerer C.F., IEEE Transactions on Software Engineering, Vol. 17, No. 12, Dec. 1991, p.1284-1288

[Grec92] Taylor your own metrics environment: AMATO - a tool for the metric-engineer, Grechenig T., Biffl S., Proc. of Eurometrics 92, Brüssel, p. 287-300, Apr. 1992

[Hals77] Elements of SW-Science, Halstead, M.H., Elsevier North-Holland, 1977

[Haus87] Über das Prüfen, Messen und Bewerten von SW. Methoden und Techniken der analytischen SW-Qualitätssicherung., Hausen H. L.,Müllerburg M.; Informatik-Spektrum, Band 10, 1987

[Hon 90] Assuring SW Quality through Measurements: A Buyer's Perspective, Hon III S.E., Journal of Systems and Software, 1990, Vol. 13, p.117-130

[Itzf86] Qualitätsmaße für SW in der Praxis, Höckel H., Itzfeld W.D., ONLINE 9/86, Sept. 1986

[Kafu87] The use of software complexity metrics in software maintenance, Kafura, D.; Reddy, G.R., IEEE Transactions on Software Engineering, Vol.SE-13, No.13, p.335-43, March 1987

[Kitc88] Monitoring software development using metrics, Kitchenham B. A. UK IT 88 Conference Publication; London, UK: Inf. Eng. Directorate 1988, p. 45-8 of xix+618, Conf.:Swansea, July 1988

[Kitc89] A quantitative approach to monitoring software development, Kitchenham B. A.; Walker J. G., Software Engin. Journal, Vol.4, No.1, p.2-13., Jan.1989

[Kitc89] Software Metrics, Ince D., Measurement For Software Control and Assurance, Editors: Kitchenham B.A., Littlewood B. London, UK: Elsevier Appl. Sci. Publishers 1989, p. 27-62

[McCa76] A Complexity Measure, McCabe, T.J., IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320., Dec 1976

[McGa88] Using software metrics and measurements to improve software productivity and quality, McGarry F.E. Proc. of the Comp. Standards Conference 1988, Washington, DC: IEEE Comp. Soc. Press 1988, p. 105 of x+111

[Oman90] Design and Code Traceability Using a PDL Metrics Tool, Oman P.W., Cook C.R., Journal of Systems and Software, 1990, Vol. 12, p.189-198

[Page89] Static Code Analysis For COBOL Development: The Advantages of An Automated Programming Support Tool, Page D.R., Unisphere 8, 12: 64-66, Mar 1989

[Paul92] Best practices of software metrics, Dan Paulish, Tutorial 3, European conference on quantitative evaluation of software and systems, Proc. of Eurometrics 92, Brüssel, 1992

[Prat84] An Axiomatic Theory of SW Complexity Measure, Prather, R.E., Computer Journal, Vol. 27, No. 4, pp. 340-347, Nov 1984

[Rech86] Ein neues Maß für die softwaretechnische Komplexität von Programmen, Rechenberg P., Informatik Forschung und Entwicklung 1; p. 26-57, 1986

[Redm90] Software Metrics - A User's Perspective, Redmond J.A., Ah-Chuen R., Journal of Systems and Software, 1990, Vol. 13, p.97-110

[Romb84] SW-design metrics for maintenance, Rombach H.D., Proc. 9th Annu. SE Workshop, NASA Godard, pp. 100-134, Nov. 1984

[Romb87] Quantitative SW-Qualitätssicherung. Eine Methode zur Definition und Nutzung geeigneter Maße., Rombach H.D.,Basili V.R., Informatik-Spektrum, Band 10, 1987

[Romb89] Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL, Rombach H.D., Ulery B. T., Proc. of Conf. on SW Maintenance 1989, Miami FL, p.50-57, October 1989

[Sama91] A Study of Software Metrics, Samadzadeh M.H., Nandakumar K., J. Systems Software, 16; p. 229-234, 1991

[Schn92] Methodology For Validating Software Metrics, Schneidewind N.F., IEEE Transactions on Software Engineering, Vol. 18, No. 5, May 1992, p.410-422

[Sher88] Computer software development: quality attributes, measurements, and metrics, Sherif, Y.S.; Ng, E.; Steinbacher, J. Naval Research Logistics, Vol.35, No.3, p.425-36, June 1988

[Sieg92] Why we need checks and balances to assure quality, Siegel Stan, IEEE Software, Jan.1992

[Whal90] SW Metrics and Plagiarism Detection, Whale G., Journal of Systems and Software, Vol. 13, p.131-138, 1990

Management Report:
Metrics to Evaluate the Quality of Your Software

presented to the
Pacific Northwest Software Quality Conference
October 20-21, 1993

by

Michael Strange
Manager, Software Quality Practice
KPMG Peat Marwick
345 Park Avenue
New York, NY 10154
(212) 872-6401

Keywords: Software Quality, Quality Metrics, Quality Assurance, Quality Evaluation, Quality Management

Michael Strange is a Manager in the KPMG Peat Marwick Software Quality Practice. He received a BS in Computer Science from M.I.T. and has assisted clients in the evaluation of software quality since 1983. He is also an active member of the IEEE and ACM. This paper presents a series of software quality metrics which have evolved through client experiences in the Software Quality Practice and related research efforts.

Management Report:
Metrics to Evaluate the Quality of Your Software

Abstract

Information regarding the quality of developed software is an important input to many management decisions. Our experience has shown that an increasing number of senior management personnel are interested in software quality. However, the trendy world of software development makes software quality analysis very difficult.

This paper presents a series of software quality metrics designed for the senior executive. The metrics focus on three key software quality indicators: project management, software design, and software testing. While the metrics provide much insight into the quality of software, they should be interpreted based on the specific project under consideration. The metrics are designed to be as easy to prepare and interpret as possible, while maintaining the critical information necessary to draw software quality conclusions.

The metrics focus on projects which have not yet been implemented. The project management metrics provide insight into the overall control of the project throughout the various phases. The design documentation metrics allow for interpretation of software quality during and immediately following the preparation of the detailed design document. The software testing metrics are specific to the testing process itself.

These metrics should provide senior management with a relatively easy-to-understand criteria for the evaluation of software and a window into the complex interpretation of software quality.

Introduction

The senior management of U.S. corporations have struggled since the inception of the first automated computer with the quality of their information systems. With the current information revolution well underway, many U.S. companies have utilized information systems as a key component of their corporate structure and therefore find themselves dependent on these systems for survival. In fact, for the first time in history, information systems are being used to provide strategic advantage in highly competitive markets.

This reliance on systems has increased the scrutiny with which Management evaluates the effectiveness of software systems. For example, many organizations are carefully justifying all software projects through the careful projection of expected benefits. However, very few evaluate the effectiveness of the final product based on these established expectations. User satisfaction is the most common measure of initial system effectiveness.

A genuine desire exists throughout the software industry to evaluate and improve the quality of software systems. However, very few companies possess a concrete basis for the evaluation of quality [11]. Therefore, Senior Management finds it very difficult or impossible to answer the following critical business questions:

1. How much should we spend on Information Systems?
2. Should we write custom software or purchase a packaged solution?
3. How much "risk" are we incurring through the use of our current systems?
4. When will our current systems need to be replaced?
5. Why do changes to our systems take so long?
6. If we think the quality of our systems is low, what should we do?
7. What skills are necessary to design and develop quality software systems?

To answer these questions effectively, we must understand the quality of the software while still in development. This paper presents a systematic categorization of software metrics which provide an understanding of the quality of software systems not yet implemented. These metrics have evolved through real-life experiences of the KPMG Peat Marwick Software Quality Practice.

Metrics have been chosen based on the complexity of the measurement and time necessary to administer the evaluation technique. For example, a software evaluation metric based on an understanding of the number of "software modules" might be more appropriate than one based on a detailed Function Point analysis due to the time-consuming nature of Function Point analysis for large software systems. The metrics presented here are divided into three categories:

1. Project Management
2. Design Documentation
3. Software Testing

Project Management

Project management is the act of controlling the scope and effort of the software project. The control of the design and development efforts is a key factor affecting the quality of the resulting software product. Poor project management often leads to poor software quality [11]. Since project management defects can be more easily corrected than technical defects, senior management is obliged to react to known project management defects or limitations by replacing or augmenting the project leadership.

In order to accomplish this, senior management must be aware of project management problems. Due to the personal, territorial, or political nature which often pervades project management assessments, such an analysis must be accomplished through the careful, diligent use of project management evaluation metrics. Senior management should remember that "you cannot manage what you cannot measure." [13]

Such a series of metrics are presented here to evaluate the quality of the project management function. The metrics take three forms:

1. a checklist of project management quality criteria
2. analysis of the breakdown of project effort
3. analysis of submitted change requests

The first metric, a quality analysis checklist, contains criteria, each weighted based on its impact on the quality of the resulting software [7, 11]. Senior management should collaborate with the project leadership to complete the worksheet. While the weighting factors (1, 2 or 3) assigned to each criteria are presented on the checklist, they should not be considered when completing the evaluation. The resulting assessment provides a general understanding of the quality of the project management function. All categories with "poor" ratings represent areas of potential improvement. Senior management should note that this evaluation is most valuable during the development of software systems.

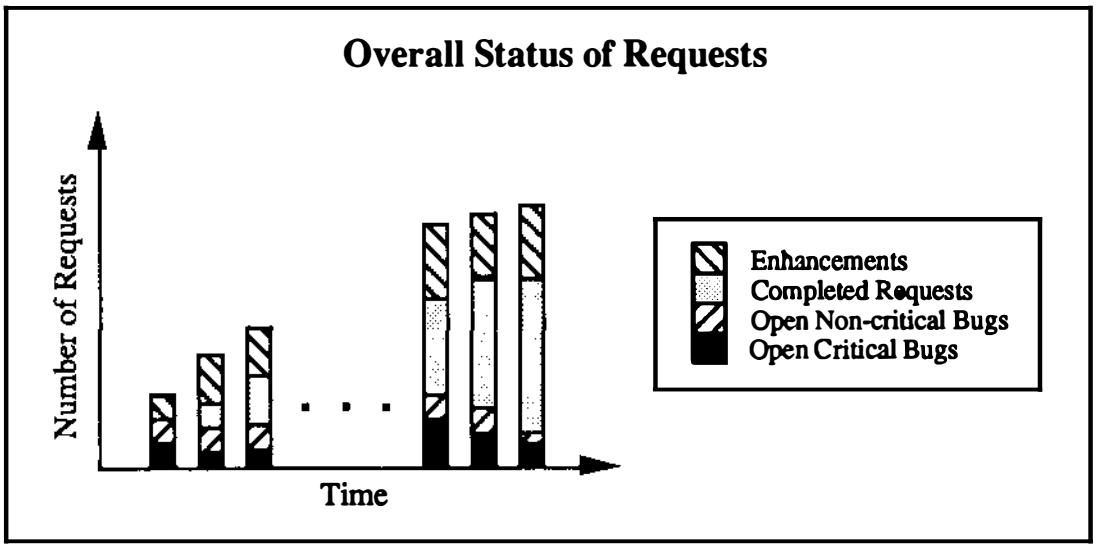
The second metric, "effort by phase", describes the amount of effort (engineering months) for the various phases of the project. While this metric clearly varies based on the technical platform, it often provides excellent insight into misdirected effort which detracts from software quality. Specifically, project managers are often inclined to begin the development phase as soon as possible. Numerous studies have shown that premature initiation of the development phase (before the detailed design is complete) severely impacts the quality of the resulting software product and often results in significant cost or schedule over-runs [10].

The project team should carefully track the engineering effort and calendar time spent on the following activities. The collected information should be compared to the metrics provided and significant deviations noted. The metrics have been compiled from published research efforts, combined with the results of the KPMG Peat Marwick Software Quality Survey [7, 8, 11].

The evaluation of the results should highlight inadequate focus on project management, design, requirements or specifications. Inadequate effort in these areas often manifests itself as inadequate detailed designs or poorly organized development effort [3, 4].

Project Phase	% effort
Initial Project Planning, Estimation, and Justification	1-3
Project Management (supervision throughout the project)	2-8
Requirements and Specifications	14-20
Design	10-16
Development (including unit testing)	35-45
System Testing	16-23
Conversion and Implementation	2-5

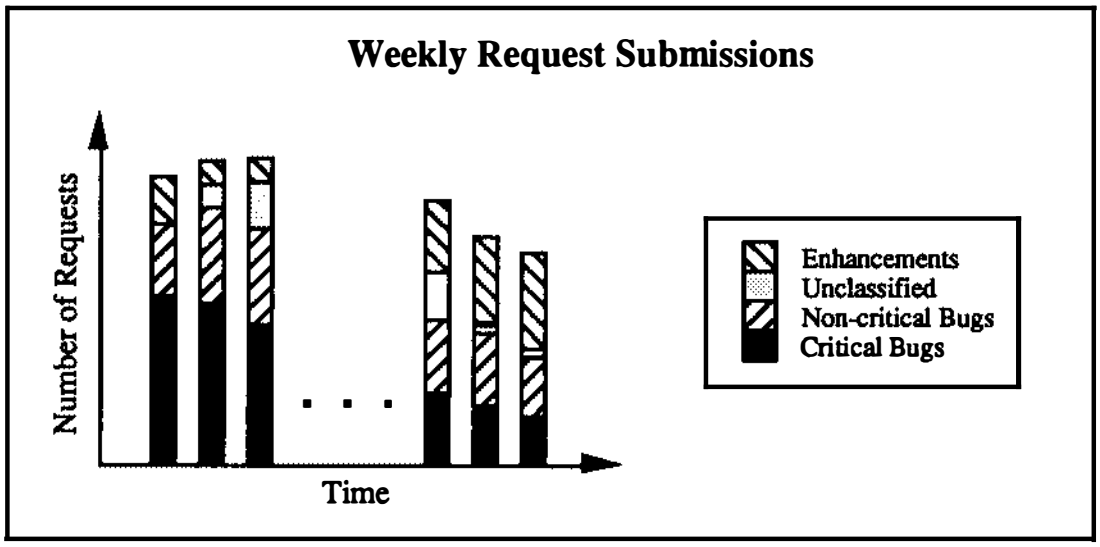
The third metric, "response to requests", analyzes the ability of the project team to receive, analyze, and respond to incoming change requests. In our experience, most project teams produce status briefings every two to four weeks during development, with the frequency increasing as the project progresses. The project team response to incoming requests and bug reports should be part of these reports. To accomplish this, two separate graphs should be prepared and presented by the project manager on a regular basis to augment the regular status reports. Both graphs have been shown to be most valuable when presented as bar charts, occasionally stacked. The first recommended graph is:



This type of graph presents the cumulative status of all change requests since the system design was completed. The total number of requests will naturally increase throughout the project. However, the number of outstanding critical and non-critical requests should decrease as the project continues.

This graph has also been used during pre-implementation evaluations. The number of critical outstanding bugs should decrease significantly (possibly to zero) prior to implementation. The number of outstanding non-critical bugs should also decrease significantly prior to implementation. Constant or increasing reports of critical software bugs is an indicator of software quality problems, inadequate testing, or a poorly controlled request response process. An inability to produce the above graph may indicate a poorly controlled modification request process. An inability to clearly define a starting point for the collection of such data may demonstrate a poorly controlled design process, in which the user community did not perceive a clear definition of the system design.

The example project depicted in the above graph demonstrates a well-controlled modification request and categorization process, based on the ability to categorize all requests clearly, including an ability to describe numerous requests as "enhancements". In addition, the above project has recently completed a successful testing period, during which several critical and non-critical problems were reported. These bug reports are being closed in a timely manner.



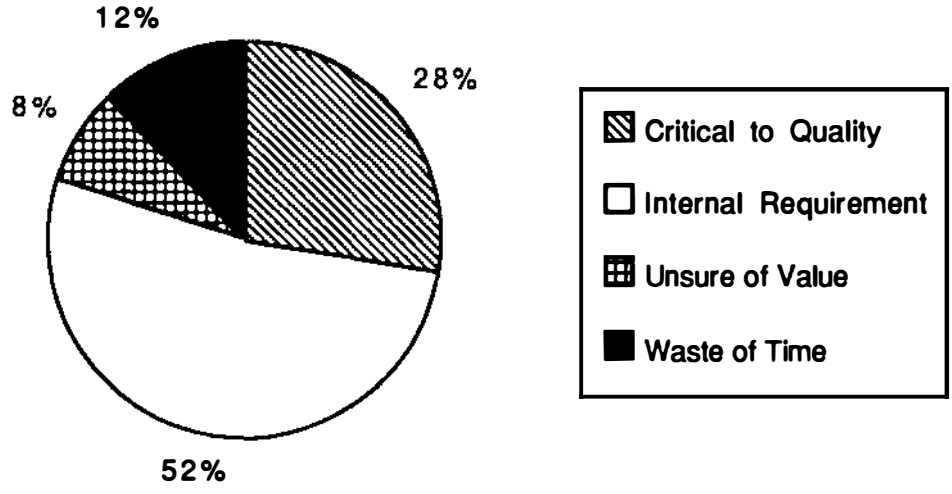
The above graph clearly depicts the status of all submitted requests. Progress toward the resolution of known problems should be accompanied by a decrease in the overall number of problems reported, specifically the critical bugs. An increase in the number of reported critical bugs during early stages of testing is normal and demonstrates the desired results of system testing -- the identification and elimination of critical software defects. However, reports of significant critical bugs during the final stages of system testing may demonstrate poor software quality.

In addition, the reviewer must remember that the quantity and significance of reported bugs is not the onl indicator of testing progress or software quality. The extent and depth of testing must be coupled with the results presented above to form a more complete impression of the quality of software testing. Despite the complexities of the software testing process, evaluation of testing results remains the most common and practical method for evaluating overall software quality.

Design Documentation

Experience, confirmed by the results of the Software Quality Survey [11], demonstrates that the quality of software design documentation is a key factor affecting the quality of the resulting software. In addition, recent research has shown that 50 to 75% of all design errors can be found through inspections. [2, 6, 14] Therefore, senior management should be able to investigate the quality of the design documentation, as a means of predicting the quality of the product or identifying problems. Since many software development personnel consider the careful design of software to be fruitless, external metrics to evaluate the quality of the documentation are needed.

The results of a recent informal study of software design and development professionals, presented below, demonstrate a fragmented, inconsistent view regarding the value of software design. This study informally asked over 100 software developers a series of questions regarding their experiences and opinions regarding meticulous software design. Those questioned were primarily development personnel with five to ten years of software development experience in commercial business. The responses reflect their opinions regarding the importance of software design on overall software quality and are limited to classic development projects (not object-oriented or assisted by CASE tools).



This survey suggests that the majority of software practitioners (52%) prepare design documentation solely to satisfy an internal requirement. It further suggests that approximately 20% of the practitioners questioned perceived little or no value in software design documentation, with 12% considering the documentation of a software design to be a "waste of time". The most significant statistic derived from this study is that only 28% of the practitioners considered the quality of software design documentation to be critical to the quality of the software product.

In this environment, software quality evaluators must be armed with specific criteria with which to evaluate the quality of the design documentation. One must remember that the natural tendency of many software development practitioners will be to "dive right into" the development of the application, believing that "the sooner we begin, the sooner we finish". It has been proven, many times, that this does NOT apply to software [7, 8, 9, 10]. In fact, just the opposite. If the project team does not carefully plan and design the application before beginning the development work, the results may be disastrous. Our experience has shown that the premature initiation of software development generally yields three results:

1. Fastest possible completion of the first development milestone
2. Significant delays in all subsequent milestones
3. In the end, a failed implementation or, at best, low software quality

Therefore, our software quality metric program addresses the question of "hasty initiation of development". Such premature development generally results in poor software quality. Indicators of premature development include:

Leading Indicators

1. Justification for informal design documentation based on the pressure to meet the first development milestone.
2. Inability on the part of the project team to describe the extent to which certain modules within the application will be used by multiple calling modules.
3. Beginning the development of application screens before all processing logic (including calculations, lookups, presentation, sorting, etc.) is understood.
4. No formal, written design documentation.
5. No user agreement to the detailed design, including a definition of all computations, lookups, presentation details, reporting requirements, and menu structures.

Trailing Indicators

1. Effort estimates for individuals modules which are wrong by 200% or more.
2. Significant re-writes for numerous application modules (in areas where no major design change has occurred).
3. No project team ability to control the impact of user change requests during development, apparently due to poor formalization of design documents.

In addition to the analysis regarding the preparation for development, this metrics program considers the quality of the software design documentation itself. When considering the quality of such documentation, the evaluator must remember that the formality with which it is created and delivered often varies between projects. Appropriate thoroughness depends on the size of the project, the tools to be used, and the ability of the user community to sustain the effort necessary to complete a detailed software design process.

The following criteria provide guidance with which to evaluate the quality of software design documentation. The criteria vary only with the size of the software project, and should therefore be used as general guidelines to be tailored to each specific situation.

	Small	Medium	Large
# Man-months	< 6	6-24	24+
Definition of scope	Informal	Written	Formal
High-level design	Informal	Written	Formal
Detailed Design	None	Informal	Written
User "signoff"	None	Informal	Formal
Reusable Modules	No Planning	Modules Listed	Detailed Design
Scope Creep Control*	None	Informal	Design Frozen

* NOTE: Scope Creep refers to an effect, usually due to poor management control of user requests, which allows a project to slowly expand in scope and functionality, eventually jeopardizing project deadlines.

Software Testing

Software testing is the most visible, quantifiable tool for software quality assurance. In addition, the effects of well-controlled testing can be significant. The integrated testing of applications is often the only opportunity to detect and correct the serious defects which cause failed implementations or poor results following implementation. Numerous studies have shown that the correction of defects following implementation can cost up to 100 times as much as just after introduced [7, 9, 14]. Practical experience has shown that inadequate or poorly planned testing can cause any of the following effects:

1. Delivery of poor software quality
2. Substantial re-writes of modules after implementation
3. Difficulty with addition of enhancements to the system
4. Inaccurate projections of maintenance costs
5. Inappropriate confidence on the part of senior management
6. Lengthy, costly, or failed implementations
7. Unacceptable down-time of applications once in production

The potential results from poorly controlled testing are very serious. Therefore, senior management must be able to determine if a given software testing program is well controlled and whether the testing is adequate. The evaluation of the quality of software testing is divided into two distinct categories, each with different metrics:

1. Test planning
2. Test results

In order to evaluate the planning and other preparation for testing, the following metric includes a number of questions which should be posed to the Project Manager responsible for the control of testing immediately prior to testing. A "yes" for all questions would excellent preparation for testing. Such preparation is usually followed by well-controlled software testing execution. The questions are as follows:

1. Has all appropriate training been done to ensure that the testers are prepared?
2. Are a series of test cases prepared?
3. Has the expected results of each test case been documented carefully?
4. Has the coverage of the test cases been considered?
5. Has the testing goals and schedule been distributed to all testers and users?
6. Is the system ready to begin testing?
7. Has the process of reporting problems been defined and communicated?
8. Does the planned testing include all aspects of the system, including screens, reports, processing, and interfaces?
9. Are all necessary testing tools ready for use?
10. Are all testing personnel scheduled and prepared to begin testing?
11. Is the user community directly involved in testing?
12. Are all necessary data conversions completed?

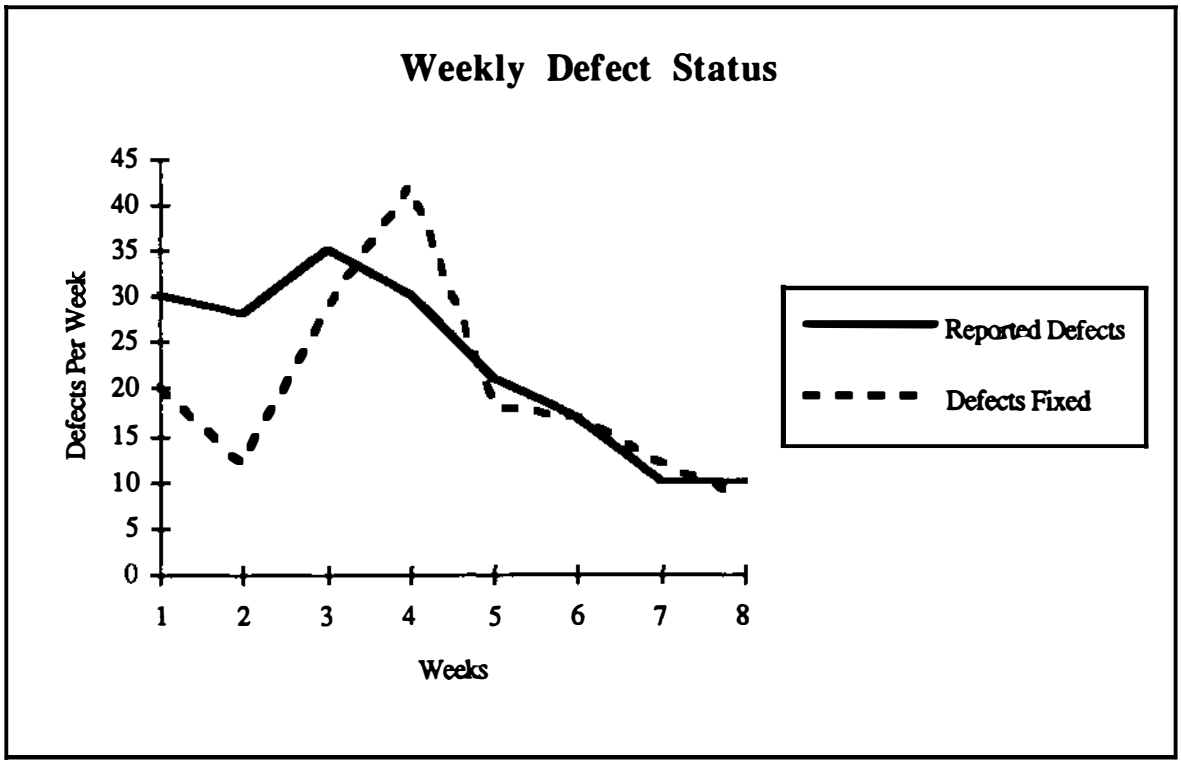
In addition, the Manager responsible for application testing should have done certain research in support of test planning. The following series of questions attempt to ascertain the detail to which test preparation was completed. While the answers themselves are not considered important for this analysis, the Manager should be able to answer the following questions:

1. How long will testing last?
2. Who will actually execute the testing?
3. What is the goal of the testing process?
4. When is testing considered complete?
5. What percentage of the system logic will be tested?
6. Will implementation decisions be impacted by the results of testing?
7. How will the status of testing be compiled?
8. How will the results be reported?

The ability to answer these questions, coupled with positive responses to the above eight questions, demonstrates a well-planned testing process. It is important to remember that a carefully planned software testing exercise involves both classic project management capabilities and technical planning expertise. The process of testing must be coordinated, planned and communicated; the details of testing must be derived from the functional and technical components of the system.

The results of testing are a key indicator of software quality, and provides great insight into the potential results of implementation and subsequent usage in production. Poor results are generally identifiable, characterized by consistent reports of significant problems or, even more demonstrative, the inability to complete the testing process due to technical difficulties. However, occasionally it is difficult to determine the impact of the results on the pending implementation. Such instances are often characterized by inconsistent results or problems which are returned by the development team as "not repeatable". Results of this nature are often disturbing in their lack of quantification.

The reader must also understand that no testing process, regardless of the preparation or positive results, will result in defect-free code [4]. It is not possible. In order to communicate the results of testing clearly to management, two graphs are presented. The first graph, depicting the reported results of testing based on the size of the application code, provides an understanding of the detailed results and the integrity of the software being tested. The second, demonstrating the number of abnormal closures for problem reports, provides valuable information regarding the consistency of the testing process and the "repeatability" of the results. The first recommended graph is:

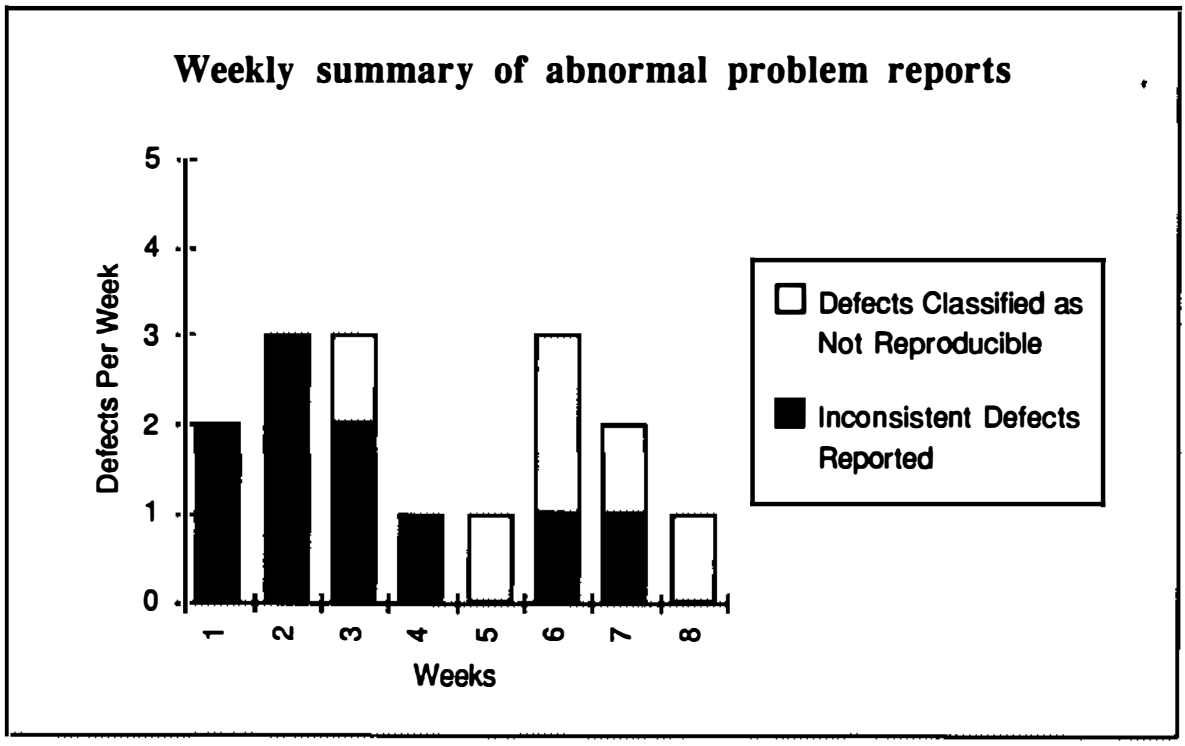


The Weekly Defect Status graph provides valuable information regarding the number of defects reported and the number fixed during the same period. The most obvious conclusion, although not the most significant, is the trend in reported defects. The reader should remember that a decrease in reported problems does not necessarily demonstrate progress.

The most significant conclusion which can be readily drawn from this graph is the combined trend in testing and fixing. If the number of reported problems is growing while the number of fixed problems shrinks, the identified problems may be increasing in significance. If both of the reported numbers is increasing during testing, the test team may be identifying an inappropriately large number of bugs. A highly effective testing effort is generally characterized by a graph which depicts the number of reported problems and the number of fixed problems steadily decreasing.

As with all metrics, the interpretation of the impact of the metric on any particular situation should be made based on the specific project environment. In this case, the interpretation of the above graph should be done based on an understanding of the project focus at the time of reporting. Specifically, the reviewer should not interpret a low number of fixed problems as a symptom of poor quality if the project team is not focused on the fixing of problems at that time.

The second graph, a weekly summary of abnormal problem reports, integrates information regarding the testing process and the response to problem reports from the development team. Unreproducible results could indicate underlying technical design problems.



The interpretation of this graph is much easier than the first - a significant number of abnormal defect resolutions is a negative indicator. The most negative conclusion from this graph would be increasing reports of inconsistent defects and constant classification of defects as unreproducible by the development team. Such reports, in our experience, always demonstrate poor underlying software quality and can often predict poor results during implementation.

Conclusion

Metrics allow us to monitor and measure software quality. Robert Grady proposes that there are three uses for software metrics, as follows [7]:

1. Project estimation and progress monitoring
2. Process improvement
3. Experimental validation of best practices

This paper proposes a series of metrics which specifically address the first of these goals. The metrics are specifically designed to provide senior management with the ability to monitor the progress and quality of software development efforts while underway. This paper presents the following metrics, divided into three distinct categories:

A. Project Management
    1. Checklist of project management quality criteria
    2. Graphical metric for analysis of the breakdown of project effort
    3. Graphical metric for the analysis of submitted change requests
B. Design Documentation
    1. Indicators of premature initiation of development
    2. Design phase quality criteria
C. Software Testing
    1. Test planning questions
    2. Test results monitoring

Strong project management will control the development of applications and allow the project to complete the desired functionality within the required timeframes. The checklist and graphical metrics presented here measure the quality of the project management function for any software development effort.

Carefully prepared design documentation is the foundation on which the application will be built. A poor, incomplete, or hasty design often results in poor software quality. The indicators and quality criteria provided will demonstrate overall design faults. Areas of weakness should be improved before beginning development. The cost of premature initiation of development is poor quality and missed deadlines.

Software testing is the focus of quality control for most organizations. The results of testing and the methods used to execute testing are the most visible, reliable indicators of software quality. Poor testing will result in the release of low quality software, usually resulting in failed implementation. The precision and ease-of-use of the testing metrics described here make them the most practical of the presented metrics.

Experience has shown that periodic monitoring of project management, careful scrutiny of the design phase, and detailed analysis of the results of testing are the three most valuable management quality criteria. The metrics provided are specifically designed for presentation to senior management, and do not replace the detailed quality assurance efforts of the project team. The metrics are simple and easy to understand, providing senior management a window into the complex analysis of software quality. This is a critical component of the information needed to answer key business questions regarding the strategic use of systems.

Bibliography

1. Basili, V., "Evaluation of Software Requirements Document by Analysis of Change Data." Proceedings of the Fifth International Conference on Software Engineering, (March 1981), pp. 314-323.

2. Boehm, B., "Industrial Software Metrics Top 10 List," *IEEE Software*, (Sept. 1987), pp. 84-85.

3. Card, D., *Measuring Software Design Quality*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

4. Demarco, T., *Controlling Software Projects*, New York: Yourdon Press, 1982.

5. Drummond, S., "Measuring Applications Development Performance," *Datamation*, Vol. 31, (1985), pp. 102-108.

6. Fagan, M. E., "Advances in Software Inspections", *IEEE Transactions on Software Engineering*, Vol SE-12, no. 7 (July 1986), pp. 744-751.

7. Grady, R., *Practical Software Metrics for Project Management and Process Improvement*, Englewood Cliffs, NJ: Prentice-Hall, 1992.

8. Grady, R., and D. Caswell, *Software Metrics: Establishing a Company-wide Program*, Englewood Cliffs, NJ: Prentice-Hall, 1987.

9. Grady, R., "The Role of Software Metrics in Managing Quality and Testing," (Jan. 30, 1990)

10. Humphrey, W., *Managing the Software Process*, Reading, MA: Addison-Wesley, 1989.

11. KPMG Peat Marwick and the Massachusetts Software Quality Council, Inc., "Software Quality Assurance Survey", 1992.

12. McCabe, T., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, no. 4, Dec. 1976, pp. 308-320.

13. Moller, K., "Increasing the Software Quality by Objectives and Residual Fault Prognosis," first E.O.Q.C. Seminar on Software Quality, Brussels, Belgium, (April 1988).

14. Myers, G., "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," Communications of ACM, Vol. 21, no. 9, (Sept. 1978), pp. 760-768.

Management Report:
Metrics to Evaluate the Quality of Your Software
Addendum

#	Criteria	Good	Average	Poor	Wt.
1	What is the average "span of control" for the project leaders, supervisors, or managers?	1 to 5	5 to 9	10+	3
2	Does the project manager impose detailed coding standards?	Yes	Limited	No	2
3	Does the project manager impose detailed standards regarding comments embedded in the source code?	Yes	Limited	No	1
4	Does the project manager impose detailed standards for the re-use of common code modules?	Yes	Limited	No	2
5	Does the project manager complete regular status reports, including progress, problems encountered, and the impact of these problems?	Yes	Limited	No	2
6	Does the project manager submit these status reports using a standard, consistent format which communicates the critical issues clearly to management?	Yes	Limited	No	1
7	Does the project manager monitor the progress of the programmers using a commonly-accepted statistical metric, such as lines of code completed per day?	Yes	Limited	No	2
8	Does the project manager monitor all change requests as they are submitted and decide whether the request is an enhancement or a correction?	Yes	Limited	No	2
9	Has the project manager clearly defined a point when the design will be "frozen", beyond which additional requests will be considered enhancements to the design?	Yes	Limited	No	3
10	Is there a clearly defined list of project goals and objectives?	Yes	Informal	No	3
11	Has the functional design for the system been documented and accepted by the user community?	Yes	Informal	No	3
12	What is the average years of practical experience for all project members?	6-10	3-5	1-2	1
13	Does the project have a defined Software Quality expert?	Yes	Informal	No	2
14	Does the project team make use of external metrics to monitor the quality of the software throughout the development effort?	Yes	Limited	No	3

1992 Metrics Research Report

Nancy Spencer
Applied Business Technology
5600 Hillcrest, #1-L
Lisle, IL 60532

Abstract

A review of the results of interviews with 28 companies that have implemented measurement. Includes the Critical Success/Failure Factors they advise.

Presenter

Nancy Spencer is the Product Manager of Applied Business Technology's metrics repository tool, METRIC MANAGER. She has an MBA from the University of Illinois, and 15 years in the IS business. She has worked in the areas of systems and programming level metrics practically since their inception and done independent consulting in measurement program implementations.

Eradicating Mistakes from Your Software Process through the *Poka Yoke* Method of Quality Assurance

by
James Tierney
Quality Assurance Manager
Microsoft
One Microsoft Way
Redmond, WA 98052-6399
(206) 936-5813
(206) 936-7329 fax
jamesti@microsoft.com

Biographical Outline

James Tierney (Bachelor of Science in Engineering, Computer Science, Cum Laude University of Connecticut 1978; Master of Private and Public Management Yale University School of Organization and Management 1988) has been working in software development and quality assurance since 1978, and has been managing software quality assurance at Microsoft for the last four years. Studying operations research at Yale, he learned about Shigeo Shingo and the *Poka Yoke* method of quality assurance. He set up the QA Department for the Multimedia Systems Group and is currently managing the Digital Communications QA Group. Microsoft is the world's largest producer of PC software and has a vital interest in maintaining the highest software quality.

Eradicating Mistakes from Your Software Process through *Poka Yoke*

Introduction

Poka Yoke (Japanese for mistake proof: a theory developed by Shigeo Shingo) is developing tools and processes to eliminate errors as close to the source as possible. I will briefly describe the development of this technique in manufacturing in Japan, talk about the general principals and application to manufacturing, and then its application in software development, including many *Poka yoke* inventions at Microsoft.

The farther along in the software process a mistake is found, the more expensive it is to correct it, often by orders of magnitude. *Poka Yoke* is the systematic practice of eradicating whole classes of errors from the software development process by locating the root cause of the errors and eliminating the potential for making that mistake. The results are less time wasted, less frustration "Why did they do this AGAIN?!", better software produced and more job satisfaction.

History--Manufacturing Process

Manufacturing has advanced far along the road to ideal quality. This is where we find the roots of the quality movement, and although software development is far different than manufacturing, selective application of quality concepts from that field into software development can be quite fruitful.

Here is a brief introduction to some of the leading lights of the quality movement.

W. Edwards Deming

Deming is widely considered the father of quality. A statistician, he introduced the concepts of statistical quality control to Japan in the 1950s. In recognition of his impact, the top quality award in Japan is called the Deming Award. Prior to Deming, people blamed poor quality on lazy or careless workers. Deming pointed out that each process will statistically have errors in some percent of the output, and to improve quality the process must be improved rather than blaming the worker.

The most important application of Deming to software is to recognize that the errors come from the process: if a certain developer or component are constantly bombarding you with errors, look at the process before blaming the worker. Some components inherently have more bugs, and the best process would take this into account.

Phil Crosby

Deming talked specifically about the factory floor: where the goal is to make many identical pieces. Phil Crosby, former VP of Quality at ITT, expanded the role of quality to include service roles. His contention that "Quality is conformance to requirements" (and be explicit about all the requirements) adds a valuable piece to the quality puzzle.

Ishikawa

Ishikawa's biggest contribution is "the customer as the next process". This is the idea that quality is not only concerned with the eventual end user, but the next person to get to work with the partial assembly. The software development application of this principle is for the developers to consider the testers as their customers and to do whatever they can to make each hand-off to test as productive as possible. Due to the iterative nature of the tester-developer relationship in software development, it turns out that the developers are the next customers for the testers work as well (this differs from the assembly line). So the testers find bugs which they send back to the developers and make the bug reports as clear and reproducible as possible is their goal.

Shigeo Shingo

Shigeo Shingo's *Poka yoke* (mistake proof) method of quality assurance ties all of these together to come up with an ideal philosophy for running software development. He looks for points in the process where mistakes are made and introduces devices to detect or correct them. This takes the blame away from the worker and makes the process capable of fewer problems (like Deming). The problem solving devices implicitly enforce requirements on the software: not just the final results but the intermediate

products as well. By addressing intermediate steps in the process, problems are found before the next worker sees them (like Ishikawa).

Shigeo Shingo was first put in charge of quality during the 1930's, as an officer in the Japanese Army. After the war, the Japanese were very interested in learning from the United States, especially in manufacturing, since the US had defeated them in part by making more and better quality war material. Shingo was an eager student of W. Edwards Deming, and started teaching Statistical Process Control for the Japanese Management Association in 1952.

Poka Yoke: Mistake Proofing the Process

In 1961 Shingo ran into a problem which led to the creation of the new method of quality control. Consulting for Yamada Electric, he found a process where occasionally a worker would forget to put a spring underneath a button in a complicated machine. The error would only be found much later and required disassembling much of the machine to fix. Statistics could not help here: no mistakes were acceptable. A simple method led to the elimination of the mistake: the springs to be put under the button were placed in a bowl. If any springs were left in the bowl, they were not under the buttons, and the mistake could be easily corrected. In fact this method eliminated that mistake from ever being made.

Shingo broke with statistics at this point. He decided that no errors were acceptable, and by using these devices mistakes could be eradicated from the process. Every time a mistake was made, the workers would find a way to prevent it from ever occurring again.

Poka Yoke Devices

Here are two lists: common *Poka yoke* devices from manufacturing and software. There are, of course, hundreds more that could be listed, but these are representative and useful.

Manufacturing

- **Guide Pins:** These prevent a piece from being put on a machine the wrong way: the guide pins only allow insertion if it is done correctly.

- **Error Detection:** if a machine detects an error, it notifies the worker by flashing a light and making a sound.

- **Limit Switches:** these prevent a drill from making the hole too deep.

- **Counters:** every time an action is performed, it is automatically counted. This prevents too few or too many.

- **Checklists**: used properly, these keep operations from being omitted.

Software

- **Quick tests:** run by developers on local builds before checking code in to source code control. They must run completely in less than fifteen minutes, hence the name quick. Some minor penalty awaits the hapless engineer who forgets to run this and breaks the build (like a sucker taped to their office door or becoming the builder until someone else breaks the build).

- **Build Verification Tests**: these are run after the software has compiled and linked, but before the testers install it. They ensure that the software has reached a certain level of functionality before spending tester time on it. More extensive than quick tests: they typically take an hour or more with different portions running on a dozen different computers at once (so a dozen CPU hours of testing take place in one hour).

- **Code Coverage**: this is a useful measure to take once the tests are complete: objectively measures what functionality has not been tested. Running this early allows time to add new tests, assuming you have reasonably good coverage to begin with.

- **Checklists:** used extensively by testers to make sure that they test everything they need to. Can be automated, but paper works just as well.

- **Full test automation:** making as much of the testing automated as possible allows you to run regression test passes on the code quickly, getting bugs back to developers before they know it, and making stress tests and configuration tests straightforward and little work.

Reducing Inventory = Reducing Untested Code

In manufacturing it is very important to reduce inventory because mistakes can hide in the inventory (more units ruined before the mistake is detected). In software development the same principle applies: software inventory hides mistakes which can surprise you and ruin your schedule if you do not fix them early. Software inventory in this context is code that has been written but not tested.

Both testers and developers need to work together to enable code to be tested earlier, reducing software inventory. The developers need to write the code in stages which can be tested before the product is complete. The tester needs to be flexible enough to test thoroughly whatever software is available. This will provide a solid baseline for the remaining software to be built upon. Using this method formal and integration testing at the end will be quicker, due both to the bugs removed earlier and development of automated regression tests which can test large portions of the software quickly.

Software Applications of *Poka Yoke*

Five generic software devices are described in the *Poka Yoke* devices section above. Most of them assume that you are already developing and using automated tests. Here are some other programs we have written to eradicate certain problems from our development process.

Stress Test verifies application behavior in low memory, low resource states

Application developers often neglect to consider the implications of system failure on their applications. Stress Test allows you to selectively limit each resource needed by the application in

order to make sure it fails in the proper manner: no loss or corruption of user data, no hanging the system. Combining this with fully automated tests can readily find many resource bugs quickly and makes these normally hard to reproduce bugs easy to reproduce.

Disk Inventory verifies all correct files are on disk, no extras

During hand-offs from development to testing, it is not unusual for extra files to appear or other files to disappear. Sometimes the changes are necessary and sometimes they are mistakes, but Disk Inventory makes sure that *everyone* knows when a file is added or deleted so that they can establish whether the new configuration is correct or a mistake. Adding this to the Build Verification Tests greatly diminishes the odds of this whole category of error hitting you.

Make gold produces perfect, virus free, golden masters

The process of making golden releases is a very involved one, requiring someone to make sure the right software is on the disk, run virus checks on it, compress it, time stamp it, and make the masters. Makegold does all this automatically, eliminating the possibility that someone will forget one of the steps. This is especially important, because this procedure is not done often and the cost of failure high, since the software gets mass produced shortly after this is run.

Glossary Manager eliminates redundancy in international testing

When doing simultaneous shipping of localized products, multiple releases are given to the internationalizers. Glossary Manager keeps track of which menus have been changed since last test which enables the testers to just check the changed text, preventing redundant checking. This allows most efficient use of the translation checkers and also saves quite a bit of time.

How to apply in your organization

"Dissatisfaction is the mother of improvement. But don't let it get wasted as complaint, channel towards solutions." --Shingo

There are several important rules when applying *poka yoke* to your process.

1. Apply Pareto's principle, the 80-20 rule: if there are a million problems bothering you, go after the most important ones first. Later on you can tackle the minor ones, but put your effort into the ones that will save you the most grief.

2. Don't be afraid to experiment. If the projected solution has only a 60% chance of success, or if it will not eradicate the problem, but only 60% of the instances of it, do it anyway! Any reduction in problems gets your group out of fire fighting mode and enables it to spend more time solving important problems.

3. Look to your test engineers for both the problems and the solutions. Get them into the *poka yoke* frame of mind and they can tell you what problems are hindering them and how they can be fixed.

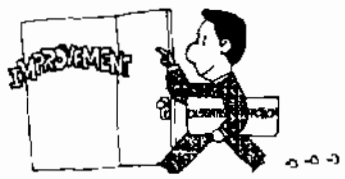
Eradicating Mistakes from Your Software Process through *Poka Yoke*

James Tierney,
QA Manager, Microsoft

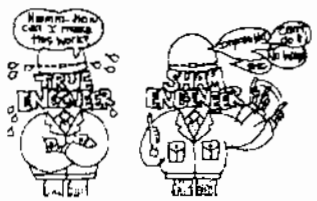
History of Quality Assurance

- Deming: statistics, not just a lazy worker
- Crosby: right the first time; conformance to spec
- Ishikawa: customer as the next process
- Shingo: Poka yoke (mistake proof) looks for points in the process where mistakes are made and introduces devices to detect or correct them

Dissatisfaction and Improvement



True Engineer vs. Sham Engineer



History of Shingo and Basis of Poka Yoke

- 1951: Head of education for Japanese management association, taught SPC
- 1961: Yamada Electric: forget to put springs under buttons. Solved with trays

Why Tolerate *Any* mistakes? Statistics Require *Some* Mistakes

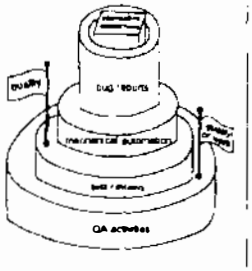
- In 1963 was called baka yoke = fool proof. But making a mistake does not make you a fool
- Design automated procedures to disallow or detect mistakes
- "Dissatisfaction is the mother of improvement. But don't let it get wasted as complaint, channel towards solutions."
  —Shingo

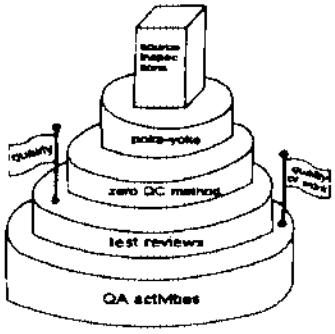
This Session Will Help You to ...

- Understand the theoretical foundations of quality
- Avoid dealing with repeated foul-ups: "Why did they do that *again*?!"
- View your software process in a constructive way (even the bad parts)
- Learn about specific methods and tools to apply to your process

Two Kinds of Forgetting

- 1. Plain forgetting (inevitable)
- 2. Forgetting that we forgot (checklists)





Poka Yoke for Software

- Quick Tests
- Build Verification Tests
- Code Coverage
- Checklists
- Full Test Automation

Reducing Inventory = Reducing Untested Code

- Linking machines to avoid inventory (pile of potential mistakes)
- Linking testers and developers to reduce amount of untested code
  - Schedule development so that test can take place as early as possible
  - Testers can use "clean room" development on well-defined components

More Software Applications

- Glossman
- DiskInf
- Makegold
- Spectool
- Stress
- HeapWalker
- Source Profiler
- DBWin

Poka Yoke Devices

- Guide pins
- Error detection
- Limit switches
- Counters
- Checklists

How to Apply in Your Organization

- "Dissatisfaction is the mother of improvement. But don't let it get wasted as complaint, channel towards solutions." — Shingo
- Pareto's Law

CHANGE MANAGEMENT IN SMALL COMPANIES

BY DONALD H. WHITE, JR.

CASCADE SOLUTIONS
4650 NW ST. HELENS ROAD
PORTLAND, OR 97210
TELEPHONE: 503-228-1300
FAX : 503-228-9707

ABSTRACT

Any changes in business rules or specifications require a modification to the existing system. Many small companies do not have the personnel or financial resources for elaborate systems to manage the changes. This paper will focus on a simple system used by a small company who has many clients that change their mind frequently and sometimes are not sure of what they want.

BIOGRAPHY

Don White's professional career began with the Logic Systems Division of Cutler-Hammer as an Application Programmer. He was a team member on a large project to control an automobile casting plant that followed a traditional software development methodology. After that project, Don has worked for several small companies in both the software development and end user environments attempting to use ad hoc software development methodologies. Those projects have included writing and or designing a variety of applications from general accounting, Data Base/Spreadsheet Conversion Utility, to Ink Dispensing System with some applications marketed internationally.

1. INTRODUCTION

The computer system at Cascade Solutions allowed the Data Processing Department a great deal of flexibility to modify software required by changing business rules and specifications. The company experienced growth and the Data Processing Department began to have a problem with managing the number of requests for changes to software. The solution was implemented with the requirements of minimum cost, reduction in complexity, and maintaining a consistent user interface. Defining the environment and reviewing the software and hardware helped the Data Processing Department analyze the problem and focus on the best solution for managing the changes. The enhanced management system had an impact on the software development at Cascade Solutions.

2. ANALYSIS OF THE PROBLEM

2.1. DEFINING THE ENVIRONMENT.

The marketing and fulfillment services at Cascade Solutions involved both manual and computer processing of jobs from clients. The jobs included projects that vary from labeling newsletters; constructing literature and product kits; data entry of business response cards; plus order entry, shipping and reporting on computer product evaluations for sales representatives. The Data Processing Department needed to capture and store data entry, print labels, or invoices and reports on information from a variety of sources. Those sources included ASCII, DBase or Paradox files from clients' diskettes, 1600 BPI Magnetic Tape, or files received over a modem. Customer or order information was entered manually into data bases by data entry operators that were on-line or on a remote PC.

The business rules for the processing of these different sources were provided by the clients. The client sometimes had not defined their requirements sufficiently or they were changed during the life cycle of the project. An example would have been a business response card. A sample was used to create Data Dictionary and Data Entry programs, but the actual card received from the client's customers had different fields. Changes were needed to be made immediately to fields in the Data Dictionary and Data Entry programs before the cards could be entered into the data base. The fields changed also affected the reports. The reports were executed manually and the last SQL command used for reports needed to be saved. The constant change to the programs, the data bases, and the SQL commands for reports or exporting of data caused a backlog of work.

2.2. REVIEW OF HARDWARE PLATFORM.

The software was developed and maintained on both an Alpha Micro AM2000 mini-computer and a 486DX PC. The Alpha Micro had a propriety operating system AMOS™ that was similar to a VAX™[1] operating system. There were about 10 Wyse RS-232 terminals connected directly to the Alpha Micro and 12 PC's connected by a propriety

network, AlphaLAN™[2]. The network allowed the PC's to function as a terminal with file transfer capabilities with the Alpha Micro or as a stand alone unit. There were two six hundred megabyte Winchester disk drives, 1600 BPI Magnetic Tape drive, Exabyte 8MM Cassette drive, interface for VCR Tape drive, various dot-matrix and laser printers attached as devices to the Alpha Micro computer. The 486DX PC had various packages used to develop run time applications that were installed on remote PC's for data entry.

The software development tools reviewed required a PC network hardware platform. A price comparison was done on a new project that required either upgrading the hardware on the Alpha Micro or purchasing a PC network and developing the software on that platform. The cost for the PC network would have been 2 1/2 times more than the cost for the Alpha Micro upgrade. The Data Processing Department had no previous experience with a PC network, so there was no guarantee that it would function to our expectations. The Alpha Micro upgrade included installing an ethernet LAN. That option would provide a chance to improve our experience with a network, but not be totally dependent on it for the success of the project.

2.3. REVIEW OF SOFTWARE DEVELOPMENT TOOLS.

The software development package and data base manager, Metropolis™[3] used on the Alpha Micro at Cascade Solutions had some good tools. There were tools for creating a data dictionary, laying out screens or reports and generated code for data entry or report programs. The tools were developed about ten years ago and lacked some of the features prevalent on the new PC based packages, but on a text based system it was better than hard coding each field for a screen or report format. The operating system, AMOS™[4] is multi-tasking and multi-user which simplified development since updated version of a program or data base did not need to be distributed as with a PC network.

Research was done on software development tools for all kinds of computers to see if anything could be used to improve the management of changes on the Alpha Micro. CASE tools seemed to have the best potential by allowing a simple graphical view of projects. The CASE tools could only be used on the PC and the diagrams could not be attached to the project on the Alpha Micro. After a brief trial on a few projects it was found to increase the time because it took longer to draw the diagram than to write the specifications. Written specifications were still required by the clients, so the CASE tool was only used for internal documentation. Configuration management tools were also reviewed, but were only available on the PC. The network on the Alpha Micro was not compatible with the configuration management tools. There had not been much of a problem with double maintenance, or shared data or simultaneous update problems[5] that would have been solved by configuration management. The CASE tools and configuration management tools had useful features, but the use of the tools would only have increased the complexity[6] of software development and not speeded up making changes to the software.

3. SOLUTIONS

3.1. IMPROVE EXISTING SYSTEM

The Data Processing Department reviewed the software products available for the Alpha Micro and the PC, plus considered the costs and affect of a new or upgraded hardware platform. The requirements of minimum cost, reduced complexity and consistent user interface kept Data Processing from trying to buy their way out of the problems with a new hardware platform.[7] Cascade Solutions decided to utilize the existing software tools and improve the change management procedures used by Data Processing and the other departments. The solution involved improving the Request Data Base, expanding the Library of Models, having a flexible standard of documentation appropriate to the project, and improving our method of capturing software metrics.

3.2. DATA BASE OF REQUESTS FOR PROJECTS

The Request Data Base was designed and implemented by the Data Processing Manager after seven years experience in the fulfillment industry at another company. There had been a need to capture the SQL commands for the many reports requested by clients. The Request Data Base contained the program name, the schedule of process dates, notes, SQL commands, and type of activity. That activity could be customized reports, labels, ASCII file import, or ASCII file export. The Request Data Base was enhanced to contain additional information to enable any report to be executed by a single command.

Each request for a client was scheduled for processing by calendar date or submitted as needed by Client Service. Any changes to the layout or to the processing of a report were noted on the request and a hard copy was submitted to Data Processing. The hard copy of the request was used to manage the changes to the programs. The example of the business response card would have notes added to the Request Data Base concerning the field changes and a hard copy printed. Next Data Processing would use that request along with a Work In Progress form to capture the time and changes made to the program or data base.

3.3. LIBRARY OF DEBUGGED MODELS.

The company that provided the accounting system on the Alpha Micro developed a generic application model with the source code. This allowed the software to have a consistent user interface through out the entire system. This reduced errors during development and testing because the same functions were used in all of the programs and only minimal changes were made for new projects. In the previous example of the business response card, there would have been fields for name and address information plus some response questions printed on the card. The model already has a data dictionary with name, address and response fields, plus data entry programs with screens and various generic reports. The development and testing time is reduced because only some of the data file fields needed to be changed along with the

screens, and import or export file layouts. Any changes to the SQL commands were made in the Request Data Base.

3.4. DOCUMENTATION APPROPRIATE TO THE PROJECT.

There was not a need to write extensive documentation on a new project because similar models were used which already had been documented. The Request Data Base had notes attached for each project of a job. A job folder was created for documentation. The folders contain the job data sheet that is updated in a data base. A hard copy was printed and inserted in the folder. Any information supplied by the client was included in the folder. It was helpful to have samples of Business Response Cards, or reports, or specified file format requirements needed to transfer data via magnetic tape, floppy diskette or EDI. It was not cost effective to store those samples electronically. There had been special note books setup for clients that have extensive file transfer requirements.

3.5. CAPTURE SOFTWARE METRICS

One set of metrics that had been recorded is the monthly total of every request processed. Information in Table 1 captured the number of requests executed, but doesn't include any design or programming work. It was used to show that a more automated approach has allowed an increased volume of requests to be processed. Another method used at Cascade to capture design or programming activity was to use the Billing Data File to record both billable and some non-billable time spent. During discussions to prepare for this paper it was decided that the method did not capture enough details of how the time was spent. A new method of capturing the actual time spent designing and implementing has been setup, a Work in Progress form. It was tested manually and will be implemented on the computer system in the future.

Month	Total Month Request Processed	Month Work Days	Average Per Day
Feb 92	393	20	19.7
Mar 92	498	22	22.4
Apr 92	697	22	31.7
May 92	717	27	34.2
Jun 92	911	22	41.1
Jul 92	782	23	34
Aug 92	847	21	40.3
Sep 92	573	23	24.9
Oct 92	857	21	40.8
Nov 92	776	21	36.9
Dec 92	922	23	40
Jan 93	799	21	38.1
Feb 93	851	20	42.6
Mar 93	1033	23	44.9
Apr 93	1257	22	57.1

TABLE 1

4. IMPACT ON SOFTWARE DEVELOPMENT

The impact of the change management system on software development has enabled the Data Processing Department to accomplish a three person work load with just a staff of two. The research done in preparation for this paper has helped the Data Processing Department to improve managing changes. It has helped reduce costs by narrowing the focus to the place where improvement would be the most beneficial instead of purchasing software and hardware that would have only increase the complexity of the problem. A mixture of manual and automated procedures was used. The Work in Progress form will enable us to better prioritize the requests and capture the total time required for the project. It is easy for professional software developers to become enamored with automated solutions that are not cost effective for small companies, but impressive to discuss at conferences. The best evaluation is one that was provided by the clients.

In December of 1992 a report card was given to all of Cascade Solution's clients. They were asked to give the service provided by the entire company an evaluation. The Data Processing Department was rated under three categories of overall, responsiveness to timely reporting, and program changes. The rating system was a range of 1 through 10 with 10 being designated as the highest. The clients that responded to the report card gave the Data Processing Department a 10 in all three categories. This provided feed back that the methods used to improve change management had a positive impact.

[1]VAX™, Digital Electronic Corporation.

[2]AlphaLAN™, Alphamicro Systems, Santa Ana, California.

[3]METROPOLIS™, AlphaBase, Los Angeles, California.

[4]AMOS™, Alphamicro Systems, Santa Ana, California.

[5]Wayne Babich, Software Configuration Management, Addison-Wesley Publishing Company, 1986, Pages 9 - 15.

[6]Peter G.W. Keen, Shaping the Future: Business Design through Information Technology, Harvard Business School Press, 1991, pages 98-99.

[7] Edward Yourdon, Decline & Fall of the American Programmer, Yourdon Press, pages 86 - 87.

CITE: An Integrated Environment for Automated Testing

Peter A. Vogel
Project Leader, Software Tools Development

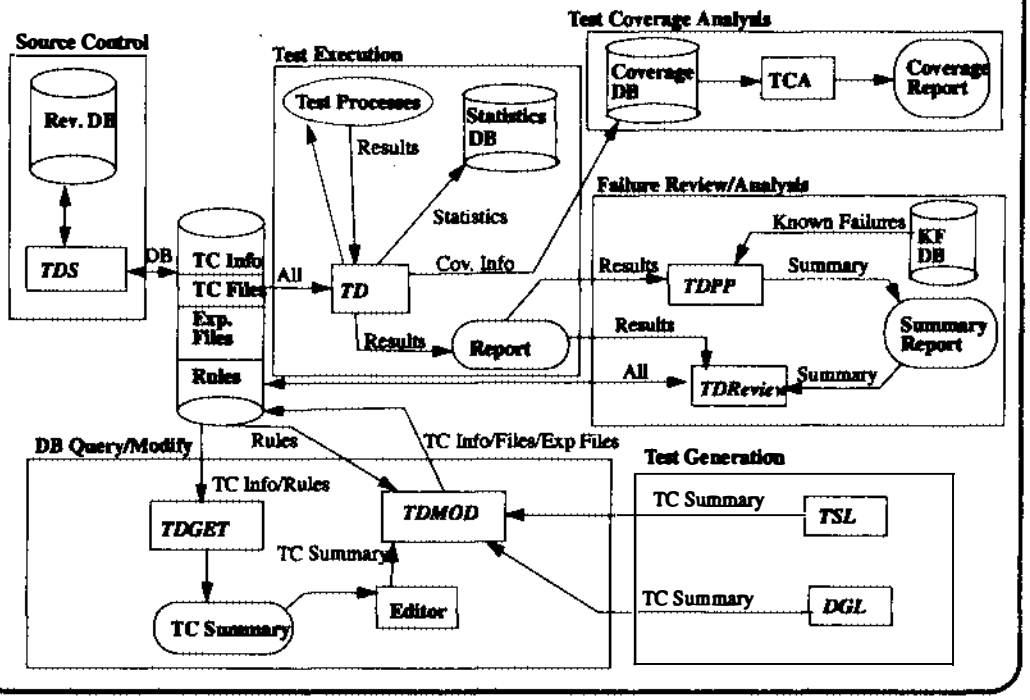
The Goal of Automated Testing

- Improved productivity.
- Improved job satisfaction.
- Higher quality.
- Faster time to market.
- Ensured reproducibility of results.
- Enhanced regression testing.

The Reality of Automated Testing

- Usually limited to record/playback.
  - Inflexible.
  - Changing results means re-recording.
  - Slow.
  - Frequently requires a remote computer to run the tests.
  - Test suites require large amounts of disk space.
- Tedious to use.
- Failures require time-consuming analysis.
- Difficult to add developer or customer tests.
- Maintenance is nightmarish.

The CITE Model



CITE — Rule-Based Testing

- Every *testcase* uses a *rule*
- Rules describe the *sequence of events* and *results verification* to be performed by a testcase.

  Example:

  > Compile a file with some set of flags. (verify rc and stderr)
  > Execute the resulting executable. (verify rc, stdout and stderr)

- Do not confuse the term *Rule* with the expert system term.
- Each testcase provides specific information to the rule it uses.

  Example:

  > File is where01.f, flags are -f90, expected rc is 0, stderr is empty
  > Execute rc is 0, stdout is empty, stdout is empty.

Rule-Based Testing

- All information regarding each testcase is stored in a database which is referenced by keys based on the rule used by the testcase.
- Advantages:
  - Easy update of expected results.
  - Storage requirements are minimized.
  - Efficient—majority of time in a test suite is execution of product.
  - Flexible—you can do almost anything in a rule.
  - Extensible—rules can be extended to give all testcases which use it additional capabilities.
  - Reusable—tests designed for one compiler can be reused to test other compilers (such as the interprocedural compiler).
  - Uniformity of execution environment allows groups to share tests.

The Rule Language

- Rules are defined in a script-like programming language which is preprocessed by the ANSI C preprocessor.
- Variables contain strings. There are three possible binding classes:
  - Export—value is defined by the user at execution time.
  - Global—value is defined by the testcase (bound at testcase definition).
  - Local—value is defined during testcase execution (a temporary).
- Type system is designed for testing. There are three possible types:
  - File— variable holds a list of files used by the test.
  - Environ—variable is to be made an environment variable for all subprocesses created for the testcase.
  - String—variable holds a string with no special meaning.

The Rule Language

- Most standard programming constructs are supported:
  - Looping (foreach)
  - Conditional (if-then-else)
  - Branch Table (switch)
- The prog statement is the heart of the language:
  - Establishes execution conditions for an event (stacksize, timeout, etc).
  - Verifies expected results.
- There are many built-in commands to facilitate usability:
  - Modify and define strings (strcat, strsub).
  - Call a system command and put results in variables (system).
  - Add information to the report file (echo, printf).

Rule vs. API-Based Testing

- Rules file provides a central point of information for a suite: if you understand the rules, you understand the suite.
- Rule-based suites are easier to maintain:

Example:

Modify all tests to invoke an optional tool after invoking the compiler.

Under API:

Edit each testcase source file (.c or .sh) to add code for invoking the tool if it is specified and to check the results of the tool.

Under Rules:

Edit the rules to add a prog statement after the prog that invokes the compiler.

Rule vs. API-Based Testing

- Rules supply the framework for a suite; once written, adding tests to a suite is trivial. Under API frameworks it isn't as easy.
  - If test is an assertion then it needs to be added to an existing test case. The appropriate source must be modified.
  - If test is a set of assertions (test case) then whole framework must be built around it.
- Management of expected results and results reporting is much easier with CITE than with API systems.
  - Automated results generation is possible with one flag to test driver.
  - API systems require the test to check appropriate results and report back to the test driver. Automatic generation of results is possible only if the test scripts allow it.

Test Suite Structure

- Suite structure duplicates structure of UNIX directory structure.
  - Each *suite* (directory) may have *subsuites* (subdirectories).
  - Suites which have no subsuites (leaves) are *basesuites*.
- Each *basesuite* contains a *database* and the *Rules* file (frequently a link).
- CITE tools can operate on any level of a suite hierarchy.

Database Query/Modify

- Tools are provided to:
  - Examine all information about testcases in a suite.
  - Modify testcases.
  - Display selected information about testcases in a suite.
  - Remove testcases.
  - Merge two test suites.
  - Browse a suite hierarchy and select testcases upon which to operate.
  - Execute tests and optionally update expected results.

Test Suite Security

- Because tests can be designated to run as any user (including root), security of the suites is extremely important.
- CITE has a hierarchy of execution privileges:
  - Execute any test.
  - Execute tests that suid only to root.
  - Execute tests that do not suid to root.
  - Execute tests that do not suid at all.
- There is one level of privilege for modification of tests, you either have permission to modify tests, or you do not.
- Sources in the test suite are protected from modification with standard permission bits.

Other Tools

- Rules are extremely flexible and powerful, but some things are better done outside the rule.
- Auxiliary tools augment the test process for "typical" tasks.
  - Automatic generation of testcases for certain classes of testing tasks (dgl and tsl).
  - Statement-level coverage analysis (tca).
  - Branch-level coverage analysis (Marick's gct).
  - Signal a process some number of seconds after starting (sig).
  - Verify certain aspects of the environment.
  - "Intelligent" diff of files (spiff with regular expressions).
  - Test suite revision control (tds).
  - Automatically generate test design from suite. (mtd)

Test Lists

- Tests within a suite can be grouped into sets called *lists*.
- Each list is a subset of the tests in the suite.
- All tools can operate on a subset of the tests in a suite specified by a *list expression* (a set notation expression).
  - Union of two lists.
  - Intersection of two lists.
  - Complement of a list.
  - Difference of two lists.
- Each suite can have any number of lists defined, with any name.

Test Lists

- Certain lists have pre-defined meaning:
  - FAIL—testcases in this list are not executed by default.
  - DANGEROUS—testcases in this list are also not executed by default.
  - CRITICAL—testcases in this list which fail cause the execution of the basesuite containing that testcase to end. No further tests in the basesuite are attempted.
- Lists can be used to provide a "virtual" view of a test suite.

Example:

> Certain tests within the procedural compiler suites do not apply to the interprocedural compiler. The interprocedural compiler suite contains links to the procedural compiler suites, but has a default list expression defined to prevent execution of tests that do not apply.

Test Lists

- Lists can also be used to correlate source code modules with tests that have uncovered a bug in that module.
- Lists can be used to prevent execution of tests that should not be run under certain conditions.

Example:

> IEEE list defines tests which require IEEE hardware to run.

Test Statistics

- CITE has the ability to maintain a statistics database containing "important" statistics about test case execution:
  - When and on what system execution happened.
  - CPU and wallclock time used by each prog.
  - Maximum datasize of each prog.
  - Result of the prog (and, by extension, the testcase).
- Tools exist to review statistics for various purposes:
  - To detect testcases that take more time (within some % tolerance) than other executions of the same test under similar execution conditions.
  - To detect testcases that were passing that now fail.
  - To review a large number of runs in a "gradebook" format.

Test Failure Analysis

- Sorely lacking in most automated test environments is an effective method of test failure analysis.
- Test failure analysis answers the following questions:
  - Was the failure caused by a bug in the product under test?
  - Can the failure be reproduced?
  - Was the failure caused by a bug in the test?
  - If the bug is fixed, does the test pass?

Test Failure Analysis

- Effective failure analysis requires:
  - Report file showing which tests failed.
  - Ability to filter the report file for known failures.
  - Access to all files produced by the test.
  - Access to actual and expected results.
  - Ability to rerun the failing tests in a suite.
  - Ability to quickly edit and update tests.
- CITE provides several tools to assist failure analysis
  - Report file filtered according to known failure (KF) files.
  - X-windows-based review tool, able to access and modify all information about a testcase and the raw and filtered report files.

Future Directions

- We intend to implement conditional known failures, allowing CITE to adapt to the conditions under which it was run and reduce the number of failures that must be analyzed.
- CITE will be enhanced to automatically make use of the parallel resources available to it.
- Exception handling will be implemented to allow a testcase to respond to the failure of events (progs).
- CITE will be enhanced to permit test source revision control at the file level, rather than the basesuite level.

Lessons Learned

- CITE (as a concept at CONVEX) is now 8 years old.
- Originally a shell script served as the driver, this helped to define the initial requirements:
  - Low execution time overhead.
  - Easy test case addition.
  - Robust error handling.
  - Allow tests to run as a specific user.

Lessons Learned

- Basic requirements drove development of TD, which survived essentially unchanged for about 6 years.
  - Rule modification corrupted the database.
  - Number of rules exploded; each basesuite had rules with the same name in many suites, each slightly different from the next.
    - FORTRAN suite had 140+ rules with unique names.
    - Some rules had over 10 permutations.
- Restricted rules modification forced "cloning" of entire suites for different products with rules modified to accomodate each product.

Lessons Learned

- Large number of upcoming products needing to share suites (RTK, C3, MPP) provided incentive for significant modification of CITE.
- Database format changed completely to allow modification of rules without corruption.
- Rule syntax changed slightly to accomodate new database
- Rules preprocessed to allow one "base" Rules file to be used for a product with differences #ifdef'ed.
- Result:
  - FORTRAN, C, and VECLIB suites all use the same base rule for about 99% of all tests.
  - This rule serves for testing all compilers on ConvexOS, RTS/rtk, TrustedOS, HP/UX and MPPOS.

Success?

- At CONVEX, CITE is used to test the OS, the OS utilities, the real-time kernel, the compilers, CXpa, CXdb, AVS, Metrics, and the C38 power software. New users of CITE are finding more and more ways for it to assist them.
- In the Development Software group (compilers and libraries), CITE is used to manage over 30,000 tests.
  - About 12,000 tests need to be run on a variety of platforms for a product qualification.
  - Approximately 100,000 total test executions are necessary for a single product qualification.
  - Product qualification takes approximately 7 days.
- Every reproducible bug reported by a customer becomes a testcase.
- ConvexOS utilities tests shared by MPPOS test group.

Improving Application Software Quality Using Rapid GUI Prototyping

By

Roger Harrison
3M Health Information Systems
575 W. Murray Boulevard
PO Box 7900
Murray, UT 84157-0900
(801) 265-4225 Office
(801) 261-3751 Fax
rogerh@code3.com

Larry Cousin
3M Health Information Systems
575 W. Murray Boulevard
PO Box 7900
Murray, UT 84157-0900
(801) 265-4565 Office
(801) 261-3751 Fax
larryc@code3.com

Abstract

Creating highly-usable systems under graphical user interface (GUI) environments using traditional tools is a time-consuming task that requires extensive training and programming. Rapid user interface prototyping tools reduce the training and effort required to produce and test experimental user interface concepts. This allows designers, including those from non-programming disciplines, to actively participate in the user interface design process and to concentrate more completely on user-centered design processes.

Roger Harrison

Roger Harrison received his B.S. degree in Electrical Engineering from Brigham Young University. He is employed as a Software Engineer for 3M Health Information Systems where he is currently doing research and design of user interfaces for advanced clinical systems. His interests include human factors, user interface design, and graphic design technologies.

Larry Cousin

Larry Cousin received his Ph.D. degree in Computer Science from Arizona State University. He is employed as a Senior Software Engineer for 3M Health Information Systems where he is currently doing research in advanced clinical fourth generation languages. He has worked extensively for over thirteen years doing software engineering for several large corporations. His research interests include software engineering, software maintenance, language theory and design, data bases, and artificial intelligence.

Introduction

During the decade of the 1980s, the concept of graphical user interfaces (GUIs) grew from the combined seeds of human factors and graphics research. This research, performed at universities and research laboratories, has developed into a commercial trend that is driving much of the new software development currently in progress.

A GUI is a visually-based workspace which graphically represents the controls and data that a user can manipulate to perform operations associated with a software application. The concept places the user at the forefront of importance relative to the application's functionality; the user's convenience and empowerment are emphasized over both the ease of programming the application and any contstraints imposed upon the application by the operating system or computer hardware.

There are many contemporary computing environments which use GUIs including the Macintosh, Microsoft Windows, Motif, Open Look, and others. In order to provide users with the power and convenience they desire, these environments provide a very rich set of capabilities. These capabilities, however, lead to a fundamental problem: designing and implementing a highly usable system under a GUI environment is a complex, subjective, and time-consuming task.

Because the user's needs and desires should receive prime emphasis in a GUI environment, the design of the user interface and the functionality it represents should also be given prime emphasis in the software development process.

It is our opinion that a sensible, systematic approach to designing, optimizing, and testing the user interface will do much to improve the quality of software by reducing opportunities for errors in the design, coding, use, and maintenance of the application. This paper outlines the rapid user interface prototyping techniques employed by engineers at 3M Health Information Systems to develop viable Motif-based graphical user interfaces for their software applications.

A Survey of User Interface Prototyping Tools

Because the need for assistance in designing the user interface in a GUI environment is almost universal, a variety of tools have been developed to aid the user interface designer. Unfortunately, most of these tools are not well-suited to rapid user interface prototyping since they require extensive knowledge and effort to produce or modify even relatively simple user interface components. Fortunately, however, there are a number of tools available which do lend themselves to rapid user interface prototyping.

The most common tools currently used for prototyping user interfaces are often the same tools used to develop them. These tools include third generation languages (3GLs) such as the C programming language and user interface management systems (UIMSs) such as UIM/X and Builder Xcessory.

The basic weakness of both 3GLs and UIMSs in prototyping user interfaces is the substantial effort and time involved in coding the user interface with them. This severely limits the number of potential user interface designs that can be explored during the prototyping stage. The constraints of typical software development projects restrict the amount of time that can be spent designing the user interface, and the time-consuming nature of using these tools for

user interface design compounds the problem. It is our experience that designing a user interface with traditional tools may severely limit both the number of design approaches that are considered and the amount of refinement that can be applied to the design that is eventually chosen. Often, a single user interface design with few, if any, refinements is the final product of using these tools. The resulting application is often confusing and difficult to use and may not even meet some basic requirements of its users.

Additionally, 3GLs and UIMSs require in-depth understanding of the application program interface (API) for the GUI environment as well as advanced programming skills which require months or years to develop. This makes them impractical for use by novices or designers from non-programming backgrounds whose input may prove to be invaluable but inaccessible with these tools.

A class of user interface prototyping tools which is gaining increasing acceptance and use is *rapid user interface prototyping tools*. While many of these tools were not designed specifically for rapid user interface prototyping, per se, they possess most, if not all, of the following characteristics [Rudolf, 1992] which make them suitable for the task:

- They allow rapid generation of basic user interface components such as windows, dialogs, buttons, and menus without programming. Generally, this is accomplished by allowing the designer to draw the visual components of the user interface.
- They allow easy modification and enhancement of previous work.
- They allow reuse of often-used composite user interface components.
- They hide unnecessary details of the GUI environment which they represent.
- They provide intrinsic high-level programmatic capabilities to allow simulation of application functionality.

It is our experience that rapid user interface prototyping tools are far superior to more traditional tools used for user interface design because they allow the user interface designer to concentrate more completely on the design of the user interface without being distracted by complex programming details.

Because the effort required to prototype design ideas is relatively small, the user interface designer can try more basic design approaches and refine one or all of them as he works toward a final user interface design. In the early stages of the user interface design, little, if any, programming is needed to demonstrate basic working prototypes of his ideas.

Because many of the details of the GUI environment are hidden, designers from non-programming disciplines can also effectively use the tool to generate design possibilities and to express their ideas. As the design progresses, the prototyping tool has the ability to simulate the functionality of the actual application which makes the prototype increasingly realistic. When the design of the user interface is completed, the prototype serves as a functioning specification of the design which can then be implemented in a single pass using a UIMS and/ or a 3GL.

There are many tools suited to rapid user interface prototyping, and new tools become available on a regular basis. A detailed survey of rapid user interface prototyping tools is beyond the scope of this paper, however it is worth briefly introducing several such tools.

- HyperCard®, from Claris Corporation, uses the metaphor of cards and stacks of cards as a way of storing information. A scripting language called HyperTalk® gives the designer programmatic capabilities. Buttons, menus, and other controls in the Macintosh GUI environment are provided to create a user interface and access the information in the card stacks.

- SuperCard™, from Silicon Beach Software, Inc., follows the stack-based metaphor of HyperCard® and uses SuperTalk™, a superset of the HyperTalk® scripting language. It provides a full complement of Macintosh GUI components for use in prototyping applications.

- Visual Basic™, from Microsoft Corporation, combines form design tools and a BASIC language programming environment for prototyping under the Microsoft Windows GUI environment.

- Visual AppBuilder™, from Novell, Inc., is an object-based application development environment that combines a forms layout tool with visual programming. It supports both Macintosh and Microsoft Windows GUIs.

- MetaCard®, from MetaCard Corporation, supports Motif-based prototyping for the UNIX X-windows environment. It also follows a stack-based metaphor and includes a HyperTalk-compatible scripting language called MetaTalk. MetaCard is the primary tool employed in the rapid user interface prototyping from which this paper is the result, and more will be said regarding MetaCard and its capabilities.

Choosing a Rapid User Interface Prototyping Tool

While few tools are designed specifically for rapid user interface prototyping, certain tools are much better suited to the task than others, particularly in light of organizational culture and methodology. While the initial goal of using a rapid prototyping tool may be to allow designers to quickly begin producing ideas, the tool must also have the depth and breadth of functionality necessary to adequately demonstrate more advanced concepts as the prototype matures.

A number of issues need to be considered when evaluating a tool for use in rapid user interface prototyping. We discuss several of these issues and give advice on how one might measure a candidate tool's performance against the criteria.

Usability of the Tool

- What is a typical learning curve for the tool?

  Basic screen layout and dialog interaction should be relatively easy to learn. A user should expect to be able to be productively doing these tasks with a few hours of experience with the tool. More complex tasks such as dynamically controlling visual

behavior will likely require more experience with the tool and can be acquired over time as they are needed.

The tool's model of the GUI environment can make a significant difference in the learning curve by reducing its complexity. For instance, the Motif GUI environment is composed of over 20 widgets (controls) each with numerous resources. MetaCard provides the primary functionality of 14 of these widgets with only four controls. In addition, these four controls can be "overloaded" to provide additional behavior that is not possible with Motif. While the MetaCard controls each have a number of properties (resources), the MetaCard environment shields the designer from having to deal with most of these properties directly.

- Is the documentation adequate?

  Good documentation is critical to being able to quickly learn and use a rapid prototyping tool. Well-organized documentation that provides pertinent information often makes the difference between success and failure in using a rapid prototyping tool. Check both the written documentation and the on-line documentation or help. See how easy it is to locate information on unfamiliar topics. Review the accuracy and completeness of the documentation.

- What other learning resources are available to the novice user?

  Books published by third parties are often available for popular rapid prototyping tools. These books can be very helpful to a novice. Occasionally vendors or third parties will also sponsor training classes on these tools. These can be an excellent way to get off to a quick start in using a prototyping tool. Other more-experienced users of a tool can often save a novice many hours of frustration by helping him to adjust to the new tool. Many companies offer free technical support to registered users of their products. Good technical support, especially via electronic mail, can be an invaluable way to overcome the initial obstacles that are almost inevitable when learning a new tool.

- How rapidly can experienced users do prototyping with the tool?

  Ultimately, this is the true usability test of a rapid user interface prototyping tool. While a good rapid user interface prototyping tool should be relatively easy to learn and should allow less-technical designers to express their ideas, it must also allow experienced users to rapidly produce, change, and enhance user interfaces. We suggest talking with other experienced users who have used the tool for purposes similar to your own to determine how well a candidate tool will measure up to this test.

Functionality of the Tool

- What are common weaknesses of rapid prototyping tools?

  It is our experience that all rapid user interface prototyping tools suffer from one or more weaknesses. In an effort to simplify their use, most rapid user interface prototyping tools hide details of the GUI environment by making assumptions regarding the way user interface components will be used. While the effects of this simplification may not be immediately apparent, they can make it impossible for the user interface designer to accomplish certain tasks.

While we have yet to find the perfect rapid user interface prototyping tool, some tools are better suited to a particular prototyping situation than others. The best tools we have seen hide many details from the beginning user but still allow flexible access to low level functionality of the tool. By understanding the capabilities needed for a given prototype and the potential limitations that candidate tools may have, the user interface designer should be able to strike a balance that fits his specific set of circumstances.

- Does the tool provide adequate building blocks for user interface prototypes?

  The tool should provide, at a minimum, the user interface components such as buttons, fields, and menus that will be used in the prototype. While most tools provide often-used basic user interface components, one may find some standard but less-frequently-used components missing from the tool's repertoire. Depending on the application, this may not be an issue, but it is worth noting.

  Flexible prototyping tools also allow the designer to use standard components in multiple, sometimes non-standard, ways. While this "component overloading" might seem to be a potential weakness, it can greatly enhance the ability of a designer to find good solutions to a particular user interface design task or to simulate complex behaviors with much less work than would otherwise be required.

- How well will the tool prototype the target GUI environment(s)?

  Most rapid user interface prototyping tools target one or more GUI environments. In some cases it may be important to use a tool that will exactly imitate the GUI environment in which the software application will eventually be implemented. In other cases, approximate behavior is sufficient.

  In connection with this, it is important to note that some tools promote but do not attempt to force any sort of compliant behavior with their target GUI environment. In these cases, it is up to the designer to know what types of behavior are acceptable in the target GUI environment to avoid prototyping things that are either stylistically incorrect or unimplementable in the final application.

- What are the trade-offs of prototyping in one GUI environment when the application is targeted for a different GUI environment?

  Prototyping a user interface in a GUI environment different from the one targeted for the final application can have several benefits. Perhaps the most obvious of these is that it may allow the use of a rapid user interface prototyping tool for a project where no tool is yet available for the target GUI. It may also allow more people to participate in the design phase of the user interface if the tool is on a platform that is more readily accessible to secondary designers and users who will test the design. The user interface designer may find his creativity sparked by working in a new or different environment which often leads to a better overall design. Because the prototype is not available in any form in the target GUI environment, it will generally have to be entirely rewritten for the target GUI environment. This has the beneficial effect of allowing the final implementation to be freed from the awkward hacks that inevitably creep into a design as it evolves.

  This approach can also have its drawbacks. The designer may find himself completely unable to simulate certain types of standard behavior which are readily available in the

target GUI environment. These incompatibilities can be caused by either software or hardware. For example, Motif uses an option menu to achieve functionality similar to that of a combo box in Microsoft Windows and the number of mouse buttons available on different systems ranges from one on the Apple Macintosh to three or more on X windows systems. Even similar but different behavior between the prototype environment and the final application environment can hide potential problems for users. Because the prototype functions as a working specification of the final system's behavior, the designer must be careful to document the places where these behavioral incompatibilities occur to ensure that the final implementation of the system is correct.

Prototyping in a second GUI environment may also require the purchase of additional hardware and software which can be quite costly both for acquisition and maintenance.

- Is the tool robust and extensible enough to allow for realistic behavior in advanced prototypes?

  As noted previously, the fact that most tools attempt to simplify the GUI environment can significantly reduce the time required to learn the basics of the tool and generate simple prototypes. As the prototype matures, however, the designer may need to simulate increasingly complex behavior. The tool's ability to adapt to these demands is critical. Facilities to manage structured data, create and modify user interface components under program control, interface to DBMSs, use existing 3GL libraries, and take advantage of operating system services such as piping data to or from other processes can all be invaluable aids to a creative designer.

  Some tools allow the integration of new user interface components such as spreadsheets, tables, and graphs which can be developed in-house or purchased. If appropriate components are readily available, they can drastically reduce the time required to prototype some types of complex behavior.

Availability of the Tool

- Is the tool readily available to all potential designers and in suitable user-testing facilities?

  This is an important consideration if design of the user interface will be done by more than one person, particularly if they are physically distant from each other. If a human-factors consultant from outside the company is involved in the project, it is important that he have access to the tool for design, critique and user-testing. Consider the effort required to "share" prototypes since it is likely that at some point the prototypes will be shown to management, support staff, documentation staff, and field sales and service personnel [Wagner, 1987].

The Application Development Process

A current software development project at 3M Health Information Systems has adopted a user-centered design philosophy using Motif-based rapid user interface prototyping to improve software quality. This software development process focuses on understanding users' needs and integrating the application's functionality with a well-designed user interface to increase the users' productivity.

User-Centered System Specification

Basic system specifications are developed to outline the fundamental purpose of the software and its areas of functionality.

The system is decomposed into sub-systems with well-defined interfaces for accessing the functionality of each part. The user interface may be part of one, several, or all sub-systems. Detailed specifications are developed for each sub-system to describe its functionality and how the sub-system uses or is used by other sub-systems.

While developing the detailed specifications, the design team turns its attention to the user in an effort to better understand the scope of the system and to gather information that will prove useful during the design of the user interface. Users are heavily involved even at this early stage of development since the user interface and the underlying functionality of the application are so closely related. The user interface is only a means of accessing underlying system functionality, yet without the proper functionality, no amount of user interface design and testing will produce a usable system [Rheingold, 1990].

During this phase of the project, the design team attempts to glean information about the user and his needs from a number of sources in an attempt to answer questions such as:

- What is the user's job?
- How does he do it now?
- How might it be done better (with fewer errors, with less effort, or in a shorter time) yet still fit within the basic constraints of his world?
- What specialized knowledge or jargon could be included as part of the system to make it more usable?
- What functionality is available in software currently employed by the user? Which functionality is most important to the user and why?
- What are limitations of software currently employed by the user? How can these best be overcome by the new system?
- What additional functionality should be included in the new system to aid the user in his work?

Throughout this process, the design team is oriented toward the users' needs rather than the solutions to those needs. The user's direct and indirect interaction with each subsystem is described in general task-oriented terms such as, "the user must be able to retrieve the desired data from a database, analyze it, and generate reports based on it." This description of each sub-system helps to define the range of the user's tasks, but no attempt is made to specify how the user will perform those tasks.

Users are visited at their work sites. Notice of the anticipated visit is generally sent to the site well in advance to allow work supervisors time to talk with their entire staff and identify key issues. During the visits, these issues are discussed at length. Users are also observed doing their actual work to help the design team better understand the users' work flow and needs.

We find that many users are solution oriented, and they will try to explain their needs in terms of a specific software feature. It is our experience, however, that several needs can

often be better met by a single more generalized solution rather than by several specific features, so the design team tries to translate each request into a task-oriented statement that describes the user's true need rather than how it will be implemented. The user's suggestions are also recorded since they may contain important clues to the true nature of the user's need or even be the best solution to the problem.

Sales, marketing and service personnel who have frequent contacts with users are also surveyed for information that will aid the design team in producing a more usable product. These surveys occur through formal and informal contact between the design team and other personnel.

If a system is currently in use, it is reviewed to ensure that the new system will provide at least the functionality that is currently available to users. Requested enhancements to the current system are scrutinized carefully since they may be indicative of larger, more general user needs than the individual requests might seem to imply.

Features available in systems with similar functionality are also considered for inclusion in the new system. This is especially true of systems that are not necessarily designed for the same type of user since they may inspire fresh new approaches to accomplishing certain tasks in the new system.

The Role of Human Factors

The integration of sound human factors principles into the design of the project cannot begin too early. Human factors experts are included in the early stages of system specification and evaluation of the users' needs and continue to assist the design team with design expertise and user testing throughout the prototyping stage and into final production of the system.

An awareness of human factors principles is fostered among the entire design team. If a human factors expert cannot be included on the design team, it becomes even more important for one or several team members to develop human-factors skills. This is accomplished in the following ways:

- Allowing the human-factors expert on the design team to introduce team members to good design principles as opportunities naturally arise throughout the design process.
- Regularly involving team members with users to help them gain a greater understanding of and appreciation for the users' needs and increase their desire to better meet those needs.
- Encouraging team members to read literature on the subject of computer-human interaction and user interface design. Much of this literature is oriented toward non-specialists and is very helpful in teaching team members basic human-factors principles that apply to their work.
- Formally educating team members through classes, tutorials and short courses on human factors. These educational opportunities are generally available to team members who live near a university or attend computer-related conferences.

Iterative Rapid User Interface Prototyping

Once the system specification is completed, the functional specification of the system, user profiles, and creative input from the design team are combined to produce a working user

interface prototype of the system using iterative rapid prototyping. The goal of rapid user interface prototyping is to quickly converge towards an optimal user interface solution as far as possible before extensive coding of the user interface takes place.

Iterative rapid user interface prototyping allows the design team to explore and demonstrate a number of approaches to the user interface with a minimum of effort and within a relatively short time. The shortened time cycle from concept to demonstrable prototype allows the design team time to consider more ideas and refine the best of those ideas to achieve a highly-usable system.

Rapid user interface prototyping is an evolutionary process designed to model, test, and refine the user's interaction with the system [Wagner, 1990] [Hemenway, 1991]. The designer begins by creating user task scenarios which describe a typical user interaction with the system. These task scenarios are then used to generate user interface ideas which are prototyped.

Developing the Prototype

The designers take a minimalistic attitude toward prototyping and focus on producing the desired behavior with as little effort and programming as possible. Often, it is possible to take shortcuts by partially implementing behaviors, by drastically reducing the number of active choices the user has at any given time, or by only simulating the effects of a user's actions.

In the beginning, the prototype may be little more than a series of sketches, either paper or electronic, linked together by real or imaginary actions. While we do use paper sketches for preliminary ideas, we tend to prefer electronic sketches made with a structured drawing tool or the screen layout facilities of the rapid prototyping tool because they are easier to modify and reuse.

If the sketches are on paper or done with a drawing tool, one can simulate user interaction by showing a sketch, explaining an action, then showing the next sketch which reflects the result of that action. This process is repeated until all of the sketches have been shown.

If the sketches are made with the rapid prototyping tool, it is possible to provide real, if limited, interaction with the prototype by linking the display of the next sketch to a user action on the current sketch such as a button press or a menu selection. We like to think of this prototype as an *interactive sketch.*

MetaCard uses a paradigm wherein each window in the system is represented by a stack containing one or more cards. We regularly exploit this paradigm by creating a series of snapshots of a window, one per card in the stack, at different stages during the program's execution. The cards are shown in succession based on user interaction. To guide the user through the succession correctly, we highlight the appropriate control for user interaction on each sketch with a green dot or some other visual cue. While this does not test a user's ability to correctly use the system without aid, the illusion of a working program is very strong. These prototypes are useful to explain and test major conceptual ideas of the user interface design.

To reduce the complexity of the prototype at this stage, we often prototype only a single linear sequence of actions which mimic the user task scenario for the prototype. This

requires almost no programming since a single command is required to move from one card to the next.

As the prototype matures we may add branches to the sequence of sketches to allow the user more freedom to explore the prototype without being locked into a fixed path of interaction. More often, however, we begin to program the *appearance* of core behaviors into the prototype so that the user can experience a full range of interaction with key components of the user interface. At this stage, our goal is simply to continue the interactive sketch paradigm by providing visual feedback that creates the illusion of the prototype doing work; generally no effort is made to actually implement the underlying functionality of this behavior unless absolutely necessary.

As the prototype matures, the body of possible user interaction is expanded until it finally represents the behavior of the entire system.

Skills for the Designer

Non-programmers with the skills to use a structured drawing program and a basic understanding of the user interface components of the GUI environment in which they are prototyping should be able to create sketches of their ideas using the rapid prototyping tool.

Often, even the more interactive prototypes can be created by designers who have limited programming experience since many typical programming responsibilities like memory management, variable declarations, and error checking are either handled automatically by the tool or are obviated due to the nature of the prototype versus a production-quality system. In addition, the languages used by many rapid prototyping tools are much less intimidating than 3GLs like C which makes them easier for novice programmers to use productively.

Usability Testing

As the design progresses, regular usability testing is necessary to validate design decisions and give feedback to the designer regarding the effectiveness of his design. Presenting users with alternative designs for evaluation can be very helpful in determining which design approaches are best. Additionally, the design should be measured against the functional specification of the system and user task scenarios to ensure that the system's basic goals will be met.

A final verification of the user interface design is done at the completion of the prototyping stage. If user input and testing has occurred at every step of the system and user interface design, this step should almost be a formality. Even so, it is important to formally verify that the design will meet the needs of the users before proceeding to final implementation.

Implementation of the Production-Quality System

At the completion of the rapid user interface prototyping stage, the design team has a working system specification for the user interface to complement the functional specifications for the system. The user interface prototype may be enhanced into a production-level system if the rapid prototyping tool will allow it, or the user interface may be implemented completely in a 3GL. Few, if any, modifications to the user interface should be required during this stage of development if the rapid prototyping stage has met its goals.

The functional portions of each subsystem are completed and integrated into the user interface scheme developed during the rapid prototyping stage to complete the system. Final system testing is performed to ensure that the system is working correctly and reliably.

Rapid User Interface Prototyping Strategies

Our experience with rapid user interface prototyping has led us to develop several strategies to increase our productivity, enhance our creativity, make design trade-offs, and overcome the weaknesses of the tools which we use. We offer these suggestions as practical advice to the designer.

- Be spartan.

  Don't do any more than is necessary to demonstrate your idea since it might be abandoned or radically altered in the next stage of the prototype. Concentrate on providing visual feedback that creates the illusion of a working system. Don't allow yourself to get bogged down with implementing real functionality, error checking, handling boundary conditions and all of the other details required for a robust production-level system.

- Think forward.

  Try to develop and use prototyping strategies that will enable you to quickly reuse or extend your work. Often using these strategies requires little or no extra effort and can pay big dividends in the future.

  Example: The first time you create a component that you may need to use regularly, make it general purpose then customize it to your application. In the future, you'll only have to customize the component when you need it.

- Approach your task creatively.

  The task of the user interface designer is to create a usable system. While many user interface design problems have neat, well-packaged solutions, there are many more for which no easy solution exists. It is up to the designer to creatively tackle his task in order to meet this challenge [Mountford, 1991].

  Force yourself out of the rigid programming approaches required during implementation of a system and look both for creative ideas for the user interface and creative ways of prototyping them. All tools have weaknesses and limitations, but creatively approaching obstacles can help you find ways to overcome them.

  Remember that your job as a prototyper isn't to create a system but to illustrate the user's interaction with the system. To quote a common expression, if you can't make it, fake it.

- Reuse everything.

  Most good rapid prototyping environments will let you reuse your own work at the very least. If an environment provides you with components that you can reuse, use them wherever possible, even if just as a temporary fill-in for a more complex object later on.

The MetaCard user environment was actually developed with itself, so one can freely reuse any object, small or large, whether it is a menu button, a visual separator, or an entire dialog window. This not only saves the time needed to create objects, but can also flatten the learning curve. When we first began using MetaCard we tried to recreate the behavior of a certain component without success. After an entire day of experimentation, we finally gave up and cloned a component in the MetaCard user interface with the desired behavior. Since then we have cloned the same component nearly a hundred times and still haven't had to learn to recreate the behavior any other way.

By reusing components already available in the MetaCard user interface, we were able to paste together the main components of an entire application in a few days. In some cases we substituted dialogs with similar purposes from the MetaCard system for dialogs in our system. Even though these did not work as we would eventually want them to, it gave us a way to quickly illustrate the system's behavior and give other design team members a rough idea of the big picture as we saw it.

Some rapid prototyping environments have many third-party components readily available. The cost for many of these is quite reasonable and their use may be easily justifiable in light of the time that they can save the user interface designer.

- Customize your environment.

  By adapting the rapid user interface prototyping tool to the task at hand, one can avoid repetitive or time-consuming tasks and concentrate on the user interface design itself.

  The MetaCard environment nicely lends itself to customization. We have made customizations as small as adding a button to a dialog to automatically resize an object and as large and creating entirely new dialogs for the MetaCard environment to simplify multi-step tasks or make up for deficiencies in the system. We also regularly write scripts (macros) to do a complex or oft-repeated tasks.

- Leverage yourself.

  Take advantage of anything which might reduce the time required to produce your prototype. We have already discussed major areas such as reusing components and simplifying tasks. Every rapid prototyping tool has strengths which can be exploited. Find these strengths and capitalize on them to reduce prototyping time or to compensate for weaknesses of the tool.

  In the multi-tasking multi-processing environment of UNIX, MetaCard supports opening pipes to processes allowing one to exploit the powerful capabilities of standard UNIX utilities such as awk and sort to quickly add functionality to the prototype. MetaCard also provides extensions to link in C libraries which allows the prototyper to reuse existing libraries or to program complex data structures that are unavailable in MetaCard.

  Do everything you can to flatten the learning curve of your tool. Consider taking formal training on the tool if it is available. Survey books that may help you learn the tool more efficiently. Develop a network of other users who can answer your questions and help you over the inevitable stumbling blocks as you learn to use the tool and with whom you can share ideas as you become more proficient.

- Look for unifying principles that bind the system together.

  Often these principles lead to a single elegant solution to several issues. The more general solution may even provide additional functionality to the system without incurring any extra effort to program or use the system [Nelson, 1990].

- Approach the design task idealistically.

  As an exercise in eliminating artificial limitations in one's thinking, imagine that the design is for an imaginary world without constraints. Develop a design approach based on this idealistic world, then try to represent or mimic it as closely as possible within the constraints of the system.

- Make the prototype realistic.

  Try to simulate the behavior of the final system wherever possible to avoid giving users unrealistic expectations. Add realistic feedback to the system. For example, build in delays to simulate time-consuming activities such as database accesses [Powell, 1990].

- Avoid redundant prototyping.

  It is unnecessary to build a new prototype for minor changes to a system. It may also be unnecessary to completely prototype parts of the system which have essentially identical behavior [Powell, 1990].

- Make simple tasks easy and complex tasks possible.

  Use the concept of progressive disclosure to hide unnecessary details from users who don't want or need to be bothered by them and still make complex functionality available to more advanced users [Norman, 1988].

- Plan for the expert user.

  Make the system adaptable to the advanced tasks and use patterns of expert users. Provide ways for them to take shortcuts and easily perform complex tasks. Make the system extensible enough to meet their anticipated needs.

- Prioritize goals and objectives of the system before prototyping begins.

  When conflicting demands are made on the design, use these priorities to assist in making sensible trade-offs.

Advantages of Rapid User Interface Prototyping for System and User Interface Design

Rapid user interface prototyping appears to provide the development team with a number of advantages in achieving software quality. Among these are:

- Shorter software development cycle.

  A major portion of the software development effort for a project can be spent in developing the user interface. As user interfaces become more complex, this situation is likely to worsen. Designing and prototyping the system before performing the final

implementation reduces the time spent reworking user interfaces in a costly 3GL environment. Once the system is adequately specified along with sub-system interaction, sub-systems that do not have user interface portions can be developed in tandem with the user interface prototype. The benefits to software testing and documentation discussed below also shorten the software development cycle.

- Higher software usability.

  The early involvement of users along with regular checks to insure that their needs will be met by the system leads to a more usable system. Potential inefficiencies in the system are usually discovered during the rapid prototyping stage which allows them to be corrected early in the software development cycle.

- Better software implementation.

  The rapid user interface prototyping approach allows the design and development team to identify potential implementation problems before the final implementation of the system begins. The prototype itself can be used as a sounding board for implementation ideas before reaching the stage of final implementation which may help the team to knowledgeably choose the most appropriate final implementation strategy.

- More thorough software testing and lower software defect rate.

  The prototype allows testers to anticipate potential problem areas of the system early in the development cycle and they can begin developing test suites before the system is completed. This allows more time for refining test suites and testing the final system before delivery.

- Better software documentation.

  Because the system's behavior is contained within the prototype, the documentation department can begin documenting the system's behavior earlier in the development cycle with less chance of major changes to the system's behavior [Norman, 1988].

  Documentation written for the usability tests can be evaluated as part of the tests and enhanced in much the same way as the prototype of the system itself. Finally, the system itself is also better documented because our approach emphasizes good documentation as part of the design stage and the prototype serves as a secondary documentation for the system's behavior.

- Easier software maintenance and enhancement.

  During the final re-implementation of the system from the user interface prototype, many of the "hacks" that inevitably crop up while iteratively designing a product can be cleanly re-implemented. This reduces the complexity of the system's implementation and makes it easier to maintain correctly.

  The rapid prototyping approach reduces arbitrary and unnecessary dependencies between the user interface and system functions, thus reducing the chance of introducing secondary errors during maintenance.

Finally, the approach emphasizes good documentation of the system's design which enhances its maintainability [Soloway, 1986].

Problems With Rapid User Interface Prototyping

The rapid user interface prototyping approach to application development has several areas where problems may arise or trade-offs may be required. Among them are:

- Limitations of rapid prototyping tools.

  The initial learning curve of rapid prototyping tools can be quite steep. It is important to anticipate the time required to become proficient with the tool as part of the development cycle.

  Problems caused by rapid user interface prototyping tools such as missing features, steep learning curves, or lack of robustness can make it difficult or even impossible to accurately prototype some system features. In some cases, a different rapid prototyping tool may alleviate these problems, in others it may be necessary to use a 3GL to accurately model some parts of the system.

- Opposition from management and engineering staff.

  Convincing management and engineering staff of the long-term benefits of rapid user interface prototyping may not be easy. Management may view extended effort in the design and prototyping stages as wasteful and want the design team to do "real" work. Engineering staff may be unwilling to change their work patterns to accommodate a new and perhaps uncomfortable software development methodology.

  Document places where rapid prototyping actually saves time, locates design flaws, increases usability, etc. to convince management of its value. Customize your development methodology to your organization's culture and needs. Encourage team members to make suggestions for improving the methodology and give them credit for their contributions.

- Difficulty transitioning from the prototype to the final system.

  Because changes and enhancements are so easily made with rapid prototyping tools, it can be tempting to prototype the system forever or to prototype things that are unnecessary. The functional specification of the system should serve as a guide to measure the completeness of the prototype.

Conclusions

Assistance in designing the user interface in a GUI environment is an almost universal need. Rapid user interface prototyping tools allow designers to concentrate more completely on their primary task of user interface design without being distracted by unrelated details.

When preparing to do rapid user interface prototyping, it is important to pick a tool well-suited to the task. The designer should consider issues of usability, functionality and availability when evaluating tools for a particular project.

While rapid user interface prototyping is not the universal answer to user interface design, it is our experience that following user-centered design processes using iterative rapid user

interface prototyping improves software quality by decreasing the development cycle, increasing the software's usability and reliability, and making the system more maintainable.

Bibliography

Hemenway, Kathleen, Jarrett Rosenberg, and Lin Brown. *Designing Graphical User Interfaces.* Human Interface Technologies Group, Sun Microsystems. 1991.

Kim, Scott. "Interdisciplinary Collaboration." *The Art of Human-Computer Interface Design.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

Dichter, Carl. "Is It Designed Right?" *UNIX Review.* June 1993.

Mountford, S. Joy. "Tools and Techniques for Creative Design," *The Art of Human-Computer Interface Design.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

Marcus, Aaron. "Designing Graphical User Interfaces." *Unix World.* August, September, October, 1990.

Nelson, Theodor Holm. "The Right Way to Think About Software Design." *The Art of Human-Computer Interface Design.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

Norman, Donald A. *The Psychology of Everyday Things.* New York: Basic Books, Inc., Publishers, 1988.

Powell, James E. *Designing User Interfaces.* San Marcos, California: Microtrend Books, 1990.

Rheingold, Howard. "An Interview with Don Norman." *The Art of Human-Computer Interface Design.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

Rudolf, Jim and Cathy Waite. "Completing the Job of Interface Design." *IEEE Software.* November, 1992.

Schneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1987.

Soloway, E., Pinto J., Letovsky, S., Littman D., and Lampert, R. "Designing Documentation to Compensate for Delocalized Plans." *Communications of the ACM.* November, 1988.

Wagner, Annette. "Prototyping: A Day in the Life of an Interface Designer." *The Art of Human-Computer Interface Design.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

Graphical User Interface Testing

by
Ellis Horowitz and Zafar Singhera
Computer Science Department
University of Southern California,
Los Angeles, CA 90089.
213-740-8056
horowitz@pollux.usc.edu

Keywords: software testing, testing GUI, testing and X Windows

Abstract:

Software testing is one of the major challenges in the software community today. Graphical user interface (GUI) testing is inherently more difficult than traditional, command line interface testing. This paper briefly describes why this is so, and gives an overview of the state of the art in GUI testing. The main focus of the paper is a presentation of the features of a new GUI testing tool, called XTester, and a methodology which can be used to create tests with the help of such a tool. XTester offers a simple mechanism for capturing and replaying test scripts. It is able to synchronize test input with window creation requiring no special effort by the tester. Bit map images can be captured and compared using several mechanisms. Special features are available to limit the size of captured images to the smallest area of interest. A complete testing language is available for the creation of customized tests. The methodology advocates that testing should be divided into layers and suggests a hierarchical approach for the organization of test suites.

Biographies of Authors:

Dr. Ellis Horowitz received his B.S. degree from Brooklyn College and his Ph.D. in Computer Science from the University of Wisconsin. He was on the faculty there and at Cornell University before assuming his present post as Professor of Computer Science and Electrical Engineering at the University of Southern California. He has also been a visiting Professor at M.I.T. and the Israel Institute of Technology (Technion). He is currently chairman of the Computer Science Department at U.S.C.

Dr. Horowitz is the author of eight books and over sixty research articles on computer science subjects ranging from data structures, algorithms, and software design to computer science education. His book "Fundamentals of Data Structures" (with S. Sahni and W.H. Freeman) has been translated into Chinese, French, Portuguese, Hungarian, and German. Dr. Horowitz has been a principal investigator on research contracts from NSF, AFOSR, ONR, and ARPA. He is a past associate editor for the journals "Communications of the ACM" and "Transactions on Mathematical Software".

Mr. Zafar Singhera received his B.S. degree in Electrical Engineering from University of Engineering and Technology, Lahore, Pakistan and his M.S. degree in Computer Engineering from Syracuse University, New York. Currently he is a graduate student in Computer Science at the University of Southern California.

1.0 Introduction

Graphical user interfaces (GUI) are recognized as a major improvement over character based user interfaces. However GUI are inherently more complex to develop and test because of their unconventional programming style and multiple asynchronous input sources. In addition the graphical nature of the objects in a GUI adds a number of dimensions to the results which need to be verified. The flexibility and variety in look-and-feel provided by the GUI toolkits of today make their testing even harder.

X windows [8] is one of the leading GUI environments. We have been building a tool to test X-based applications. This tool, called XTester, provides several substantial improvements over current tools for testing X-based user interfaces. XTester operates at X protocol level and is capable of testing any X application no matter what toolkit it is developed on. It does not require access to the source code of the application under test. Unlike most of the existing tools, it does not require a server extension for input synthesis or any custom X libraries for resolving object naming issues. The tool primarily consists of a library of functions which allow the user to either write scripts or capture and later replay user sessions. In this paper we review the current state-of-the-art in testing GUIs. We then provide an introduction to XTester and describe its key features. In the end, we present a methodology which can be used to test a graphical user interface by using a tool like XTester.

2.0 State of the Art in GUI Testing

Most of the existing GUI testing tools [1], [2], [3], [4], [5], [6], [7] primarily focus on regression testing. They provide a mechanism to capture a user session in a format which can be replayed later by the tool automatically. The captured data consists of user actions, such as keyboard and mouse clicks, and images of application objects which need to be verified during replay. Some of the major problems with these tools, which have been observed by others e.g. Kepple [3], Tillson [4], are:

- All user actions are captured in terms of absolute screen coordinates. If the location of an object is different during replay from its location during capturing, then the test fails.
- There is no mechanism to write test scripts "from scratch". One has to postpone the testing activity until the application gets to a level where it is capable of capturing test data.
- The verification mechanism relies on location and dimensions. An image comparison of a window gives a false alarm if the window is displayed at a different location or has been resized.
- The verification mechanism is insensitive to X environment resources. The pixmap comparisons might fail due to a slight difference in X environment during capture and replay.
- All the user actions during capture mode are captured, no matter if they are related to the particular application or not. This requires an identical display environment during capture and replay.
- The captured test data depends on the underlying hardware and cannot be successfully used on a different kind of display.
- Minor changes in the software might completely invalidate an existing test, and that test has to be recaptured again.

- There is no way to include information about the software and the testing assumptions in a captured test script.
- The synchronization mechanism (making sure a window exists before using it) chiefly relies on the user and is primarily limited to waiting for windows. The user has to specify explicitly when such a synchronization is required.

On the other extreme, the approach suggested by Lawrence Kepple [3] is to define a virtual toolkit on top of all existing GUI toolkits and map all the objects in a GUI into the objects in the virtual toolkit. A tester writes tests in a language, called T, which has been specially designed for the proposed virtual toolkit. The names of the widgets in a particular GUI under test are translated into T language widget names. The binding of these names to the actual widget names is done by Window Declarations which are placed in include files. This approach requires that a mapping between different names for Open Look, Motif, etc. be maintained. A "typical" X application for a commercial software product may easily have thousands of widgets. This implies several thousand name associations must be maintained. Moreover the application under test must be linked to a custom X library, specially modified to support the tool. Another disadvantage is that this approach does not provide any support for capturing and replaying of user sessions.

Our goal in building XTester was to develop a tool which provides a simple way to capture and replay user sessions. This capture and replay should be done in such a way that the generated test data do not rely on absolute object location and dimensions. The tool should also provide a programming interface so that a tester can generate test scripts manually. The created tests should refer to objects by name and should not be affected by a changed environment during replay. The tool should provide automatic synchronization during playback so that an event to an object is simulated only when the object is realized on the display. Otherwise the tool should wait for the object to appear before generating the event. The tool should also let the user capture the minimum information which is absolutely required for verification. The generated tests should successfully play back on different platforms, or under different window managers without concern.

3.0 The Xtester Prototype

XTester provides a compromise between the existing tools and Kepple's approach. It provides an environment which is equally convenient for capture/replay as well as manually written scripts. Its programming interface is standard C language and all the standard development and debugging tools for C can be used for development of test suites. Moreover tests can use the standard XLib and toolkit capabilities to get information about objects for verification and decision purposes.

In this section, we shall briefly introduce the major windows of XTester, describe their functionality and discuss related issues.

3.1 Capture Mode

Figure 1 shows the initial screen that is displayed when XTester is started in capture mode. This initial screen is intentionally designed to be small, as it usually must co-exist with the application being tested. Once the Capture button is clicked, all keystrokes and mouse clicks sent to the

application will be captured and saved in a script file. Once clicked, this button changes its name to Pause. To temporarily halt the capture of the script, click on Pause and the button changes back to Capture. The Grab and Waitfor buttons are initially grayed out. They become operative only during the Capture phase. The Script button causes a window to appear that contains the script as it is being formed. This is explained ahead. Similarly, the Model button causes a window to appear that contains a graphical view of the window hierarchy of the application. That is also explained more fully ahead.

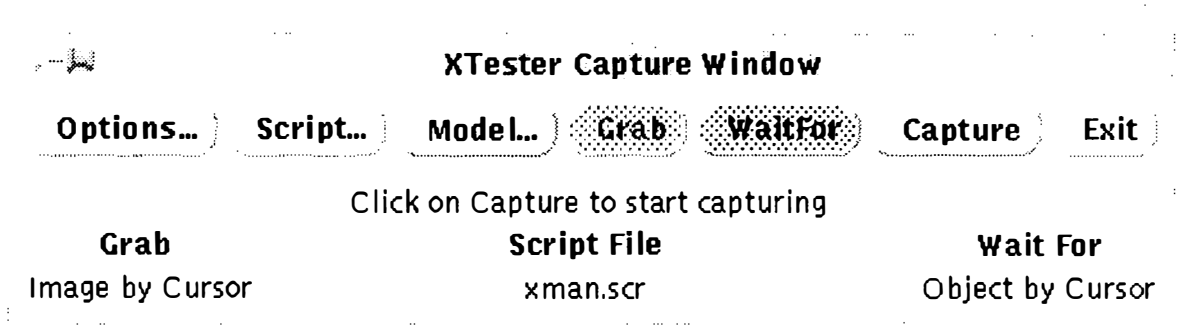


FIGURE 1. : XTester main window in capture mode

3.1.1 Capture Mode Options

There are several important sets of options that can be set during capture mode. One of these has to do with the capturing of images. During a test session it will be desirable to capture bit map images of the application, or part of the application. During replay these captured images are compared with the images generated at replay time.

Since bitmap images can often require a great deal of storage, XTester provides several mechanisms to capture the *least* amount of data for test verification. It is capable of capturing object attributes instead of images when it is feasible. It provides the capability to store and compare only checksums instead of entire images. Although a user can choose to compare the images of the topmost application window, by default XTester captures images of the smallest enclosing windows. The user can also stretch and select a rectangle for image comparison. The coordinates of such a selected area are stored with reference to the smallest enclosing window.

In Figure 2 you see the options window which appears when the Options button is clicked in the main capture window. The Capture options window is divided into two parts, an image and wait section. The Image section provides options which relate to capturing object information. Format provides an exclusive set of options so that the user can choose the format in which information should be captured. The Image option enables the capturing of pixmaps, the Attributes option activates attribute capturing, while the Checksum option instructs XTester to store only checksums instead of entire images. The Grab by option is also a set of exclusive options. It allows the user to choose the mechanism by which the object to be captured is pointed out by the user. The Cursor option in this set lets the user choose the desired object by clicking on it, as in the xwd utility. The Key option lets

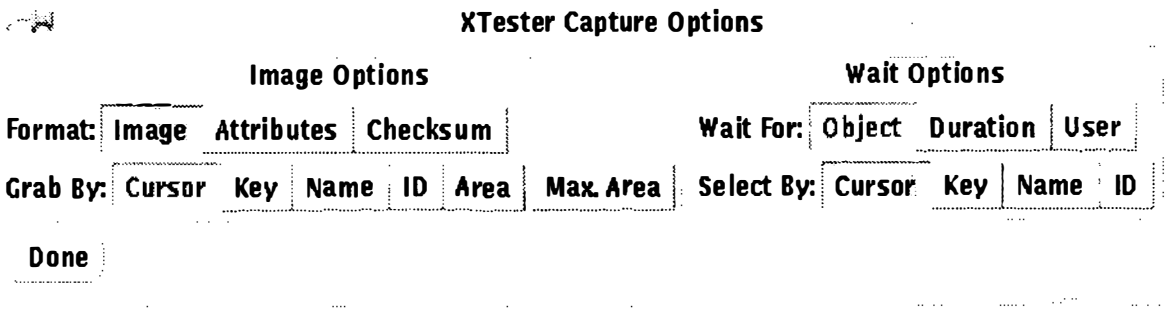


FIGURE 2. XTester capture options window

the user choose the object by first moving on the desired object and then clicking the designated key. The ID and Name options popup a dialog box and ask user to enter an object ID or Name respectively. The Area option allows the user to select a particular area in an application window, by stretching a rectangle around it. The area is captured with reference to the smallest enclosing application window. Max. Area is a toggle option which modifies the behavior of object selection. When Max. Area toggle button is on then XTester tries to capture the largest possible area or top-level window of the application, instead of the default which always captures the smallest enclosing window.

The Wait section contains the options related to WaitFor events. WaitFor is a set of exclusive options. Object, Duration or User options instructs XTester to generate script commands to wait for an object, a duration or a user response. Select by options are similar to Grab By options in the image section and are used to select a mechanism to mention the object to wait for. The options currently selected are also displayed in the main capture window so that the user knows about the current selections without requiring the options window to be on the screen.

3.1.2 Saving a Test

When a test is captured, two files are created. A .scr file (for "script") contains all of the actions, mouse movements and keyboard clicks, that occur during the test. A .img file (for "image") contains all of the images that were captured during the test. The .scr, or script file, is an ASCII file and so it is easily viewed on the screen and edited. The actual commands that appear in the file are explained in Section 3.3. The .img or image file is a binary file. It contains all of the captured images, in pixmap format or depending upon the image options selected at the time of capture.

The prefix name for each file is determined by the user. Names can be chosen to indicate the order of testing, or the content of the test. We will not say more about test administration here and reserve this topic for a later paper.

The script file is machine and environment independent. It can be moved to another platform and faithfully executed, despite differences in the cpu, monitor, or X environment upon which the test is being played. This is due to the fact that Xtester relies upon the X protocol, the lowest level of X Windows. On the other hand, the image file contains bit map images that were snapshot on a

specific monitor in a specific format. As such, these images may not be the same as new images taken on another monitor.

3.2 Replay Mode

Figure 3 shows the initial Xtester replay mode screen. Once again this screen is chosen to be small so as not to claim significant screen real estate. Buttons such as Options, Script and Model have similar interpretations as they did for the Capture window. The main operation one can do initially is to start the replay session. This is done by hitting the Replay button. The Next Step button is initially grayed out and becomes active only when running in single step mode or when a WaitEvent command, to wait for user response, is encountered in the script. None of the stored and hence replayed events are location specific. Rather all the events refer to a specific application window and the location is referenced with respect to the window origin. The tool does the basic synchronization automatically by verifying the existence of a window before any event is generated for it.

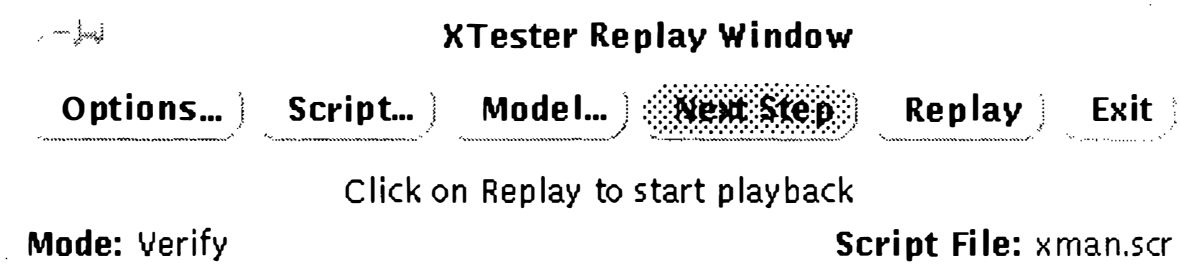


FIGURE 3. Initial XTester main window in replay mode

3.2.1 Replay Mode Options

XTester is capable of replaying scripts with or without any verification of results. It also provides a third mode of playback which considers that the results generated during replay are to be preferred, so if they differ from the stored ones XTester replaces the stored results with the current ones. XTester tries to verify a result a number of times before declaring a failure. This number can be adjusted by the user depending on the nature of the application and the environment in which it is running.

Figure 4 shows the XTester replay options window. The window is divided into two parts, Verify options and Timing options. Verify options lets the user choose the current mode of replay, i.e. whether results should be verified or not and if different, whether to log an error or replace the stored results with the new ones. The Max. Comparisons text field allows the user to specify the number of comparisons to be performed before declaring a failure. Timing options are related to replay speed of the application. The Minimum Delay field specifies time in seconds to wait before replaying the next script command. The Maximum Delay field specifies the maximum amount of time XTester

should wait for an object to appear on the screen, before giving up on it. XTester also tries to recover gracefully from failures. The Maximum Tries field specifies the number of consecutive unsuccessfully replayed commands before aborting the script. The Single Step button is a toggle button which flips between single step and continuous mode. When it is selected, it activates the Single Step button in the main replay window and instructs the user to click on that button to execute the next script command.

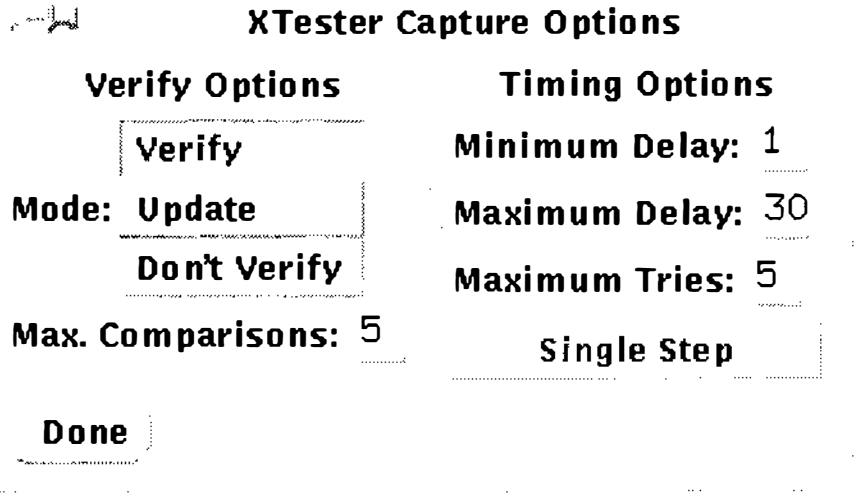


FIGURE 4. XTester replay options window

3.3 The Script Window

Figure 5 shows the script window of XTester. This window is displayed when the user clicks on the Script button in the main capture or replay window. The currently displayed script in the window was captured to test the functionality of *xman*, an X interface to the UNIX *man* utility.

The legal commands in a script file follow. The reader is advised to skip the commands on his first reading as they are not needed to understand the remainder of the paper.

`FileHeader(version,time,creatername,startCommand,ImageFile,Misc.Info)`

`FileHeader` designates the beginning of the script and contains information such as version number of Xtester, date of creation of the script, author of the script, command string to start AUT, and name of the `.img` file. A miscellaneous field is present which can be filled in by the tester for a brief description of the test or other miscellaneous information.

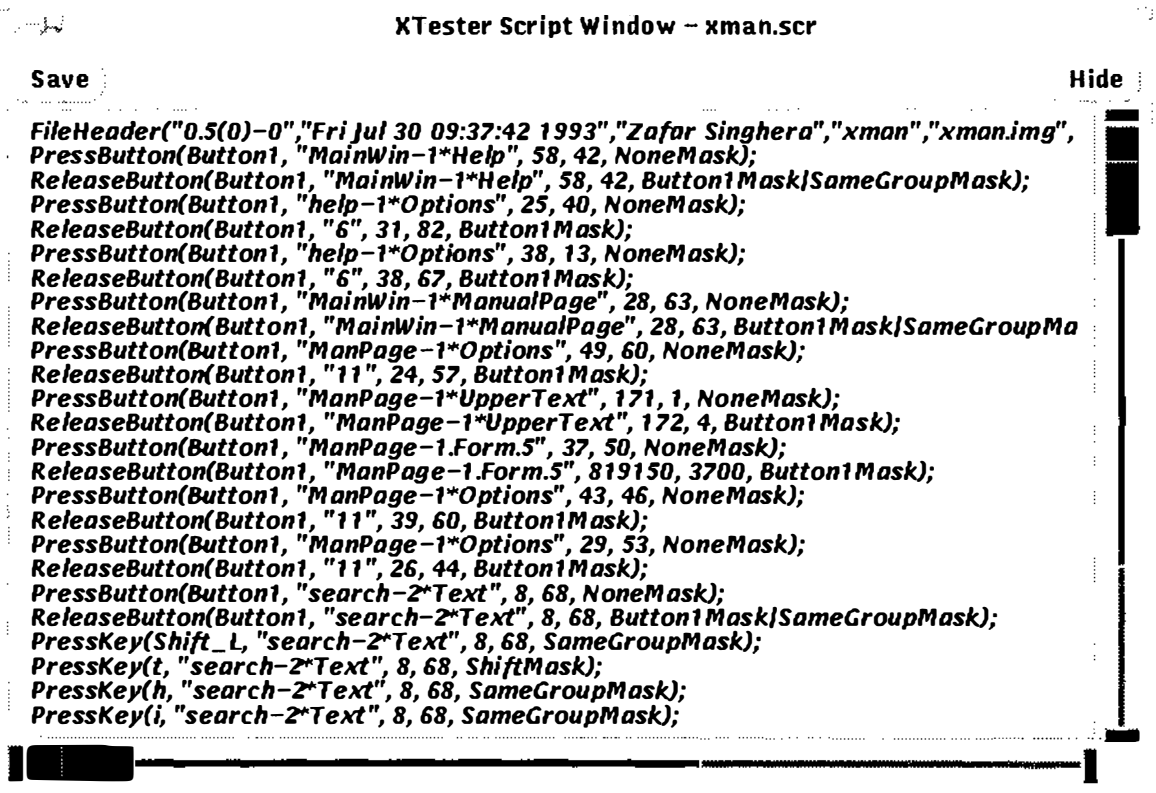


FIGURE 5. XTester Script Window

```
PressButton(button, window, x coord, y coord, state)
ReleaseButton(button, window, x coord, y coord, state)
PressKey(key, window, x coord, y coord, state)
ReleaseKey(key, window, x coord, y coord, state)
```

These commands indicate which mouse/key button was pressed/released, upon what window at what (x,y) coordinates relative to the upper left corner of the window, and state of the modifiers at the time button/key was pressed/released. State field also indicates if the event is related to the previous event or not.

```
MovePointer(detail, window, x coord, y coord, state)
EnterWindow(detail, window, x coord, y coord, state)
LeaveWindow(detail, window, x coord, y coord, state)
```

These commands represent motion related events. `Detail` field provides some event specific information (i.e. Motion was a hint or normal motion event, etc.), while the other fields are similar to the ones described for the previous commands. XTester does not capture all the motion specific events but only stores ones which are absolutely required for a successful replay.

`MapWindow(window)`

`UnmapWindow(window)`

`MapWindow` and `UnmapWindow` represents mapping or unmapping of window by the window manager or another application. Normal mapping and unmapping, done by the application under test, is not captured.

`ConfigureWindow(window, x coord, y coord, width, height)`

`ConfigureWindow` represents the movement or resizing of window by window manager or another application. `x coord`, `y coord`, `width` and `height` represent new location and size of the `window`.

`WaitEvent(detail, window, Min. Time, Max. Time)`

`WaitEvent` represents delay in the script. `Detail` field indicates the delay type and its possible types are `WaitForObject`, `WaitForDuration` and `WaitForUser`. `Min.Time` is the minimum time to wait, whether the wait condition is satisfied or not. `Max.Time` is the maximum time to wait before declaring a failure. `Window` gives the name of the object to wait for, if the detail field is `WaitForObject`.

`ObjectInfo(format, window, index)`

`ObjectInfo` verifies the object information. `Format` indicates whether the information is captured as pixmap, checksum or attributes. Window is the name of the corresponding window. `Index` represents the entry number for this information in `.img` file.

`ExecCommand(command, index)`

`ExecCommand` provides a way to verify or modify environment during playback. Command is executed and the results of the command are either stored in `.img` file, if in capture mode or compared to the already stored results, if in replay mode. If `index` is `0` then only command is executed and the results are ignored.

`Message(string)`

`Message` command prints the string in logfile. It provides the user a way to make logfile more legible.

`EndOfSession(i)`

`EndOfSession` indicates the end of the `i`th session in the currently active script.

`# Comments`

It is possible to add comments in a script. A line beginning with # (pound sign) is considered to be a comment and is ignored by the script during playback. Blank lines are also ignored.

A click is represented by adjacent occurrences of `PressButton` and `ReleaseButton` where the state field in the `ReleaseButton` event contain `SameStateMask` in addition to other information. For example, the following script commands represent a double click on `Help` button in main window of `xman`.

```
PressButton(Button1,Xman.Form.Help,40,45,NoneMask);
ReleaseButton(Button1,Xman.Form.Help,40,45,Button1Mask|SameGroupMask);
PressButton(Button1,Xman.Form.Help,40,45,SameGroupMask);
ReleaseButton(Button1,Xman.Form.Help,40,45,Button1Mask|SameGroupMask);
```

A click-and-drag is represented by a combination of `PressButton`, `MovePointer` and `ReleaseButton` commands. For example the following commands represent selection of text in `xman`'s `Help` window by click-and-drag.

```
PressButton(Button1, Help*SCWindow,99,29,NoneMask);
MovePointer(Normal,Help*SCWindow,0,0,Button1Mask|SameGroupMask);
ReleaseButton(Button1,Help*SCWindow,0,0,Button1Mask|SameGroupMask);
```

The left mouse button is initially pressed while the cursor is over window `Help*SCWindow` with offset (99,29). The cursor is then moved to window `Help*SCWindow` at location (0,0), and then the mouse button is released.

3.4 The Model Window

Figure 6 displays the model window which appears when the user clicks on the Model button in XTester's main Capture or Replay windows. This window displays the window hierarchy of the

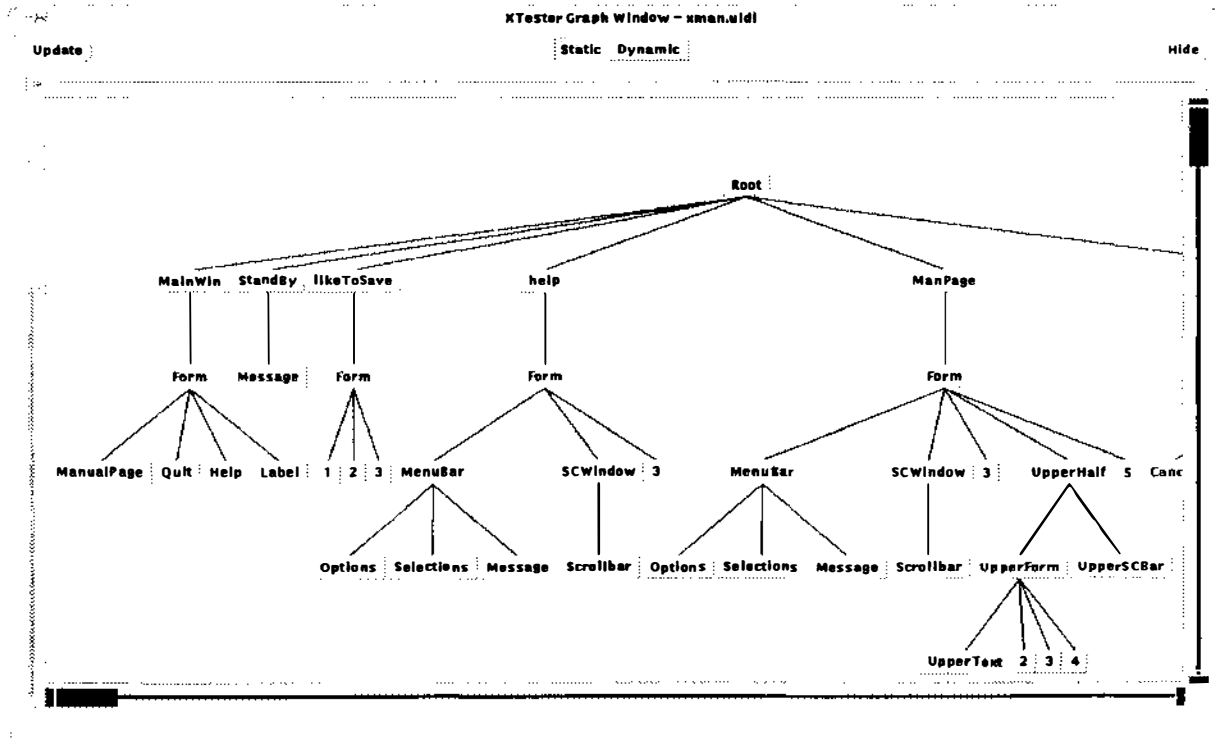


FIGURE 6. Static hierarchy of *xman*

application under test. In this paper the model window is displaying part of the window hierarchy for the application *xman.* Actually there are two window hierarchies handled by Xtester, the static and the dynamic. The user can display the required hierarchy by choosing the corresponding exclusive option from the button set in the middle of the window. The static window hierarchy is created from a UIDL (User Interface Description Language) file. It contains the complete (or a portion) of the windows, their parents and children, contained in the application program. Although XTester does not rely on the UIDL file for normal capture/replay, it is required for writing manual scripts. XTester provides the capability to build this file incrementally. Each window has a name provided by the tester. This name will appear in the script file when the object is activated.

The dynamic tree hierarchy is the one that currently exists for the application under test. This tree represents all the windows that are currently created, though not necessarily mapped. Generally the static hierarchy represents all the unique objects in the application, no matter whether they currently exist or not. Once complete, the static hierarchy never changes during a particular session or across sessions. The dynamic hierarchy represents the current population of objects in the application. Each node in the dynamic hierarchy represents an application window which is currently alive. The nodes of the dynamic hierarchy are in fact instances of the nodes of the static hierarchy and a node in the static hierarchy can have none, one or more instances at a particular time during a session.

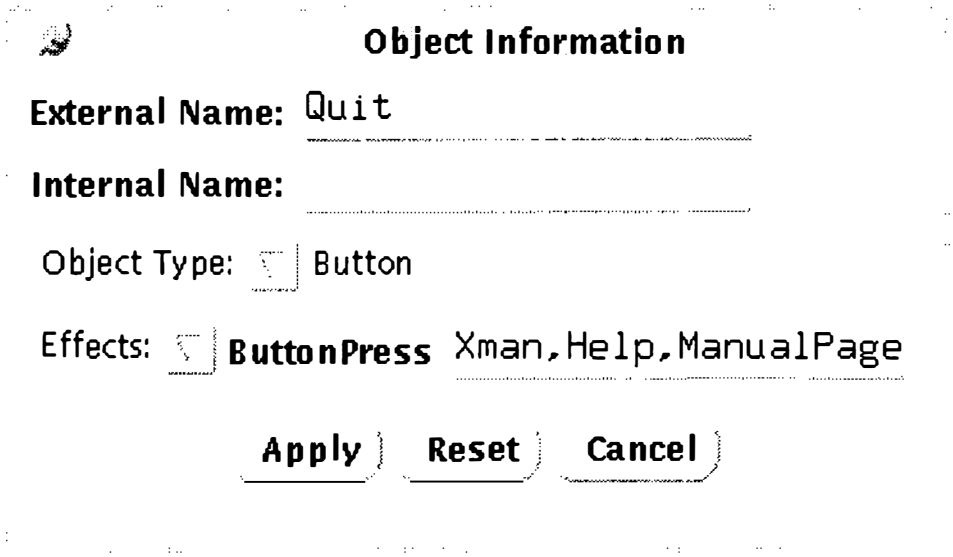


FIGURE 7. Object Information Dialog Box

When a node in the dynamic tree is clicked, it highlights the corresponding window in the application and gives information about that window in the footer area. When a node in the static hierarchy is clicked, it highlights all of the application windows which are instances of that particular node and displays their IDs in the footer area. It also pops up a dialog box, so that the user can modify the information about the node, which is stored later in a UIDL file. Figure 7 shows the dialog box. The External Name is the name given by the tester to the corresponding window. The Internal Name describes the internal name of the window, if one has been assigned by the application. This name

can also be a comma separated list of names if the application gives multiple names to the window. Object Type is a pulldown menu which allows the user to specify the object type related to the window. Object types include all the common widgets available in existing toolkits. Effects is also a popup menu which contains all the possible events which can be exercised to the object. The user can select a particular event, like ButtonPress, and then describe in the accompanying text field a comma separated list of object names which are affected when the chosen object receives such an event.

The UIDL file for an application describes a hierarchy of objects, starting with root which represents the root window of the display. Each object description is divided into three sections, Parameters, Effects and Children. All the sections and all the entries in a section are optional. The parameters section describes parameters of the object, like its internal name, full name, object type, etc. The effects section describes the potential effects of different events on the object to the other objects in the application. The children section lists all the children of the object. See [9] for further details about UIDL syntax.

3.5 Writing Customized Tests

XTester is built as a library of functions which can be used to develop tools for testing X applications. XTester provides routines to perform all levels of testing functions. The windows discussed above are generated by the FollowUser() routine of the XTester library. This routine takes over when called and depending on mode either captures or replays all user actions. On the other extreme, the XTester library provides routines to replay low level events like ButtonPress and ButtonRelease. It also provides macros to simulate a mouse click, click and drag, entering text and other similar functions. The formats of low level event simulation routines are very similar to the corresponding script commands so that a user can convert a captured script into a C program and vice versa with minimum effort. A detailed description of the programming interface of XTester can be found in [9].

4.0 A Methodology for GUI Testing

This section provides a methodology for testing an application with a graphical user interface, using XTester or a tool with similar capabilities. The methodology takes an object oriented approach for behavior verification and a layered approach for organization of test data. The methodology consists of three major steps, i.e. modelling the application under test, creating the test suite, exercising the test suite and analyzing the results. The following sections describe these steps in detail.

Figure 8 shows a window which will be used as an example during the following sections. The window contains two control areas. The upper area contains a menu button, `Edit`, and a text widget with its label. The menu entries, `Cut` and `Paste`, are used to manipulate text displayed in the text widget. The bottom control area contains a command button, `Dismiss`, which makes the window disappear from the screen.

4.1 Modelling the Application Under Test

The very first step for any testing activity is to get information about the structure and behavior of the application under test and organize this information in a useful manner. For testing the graphical

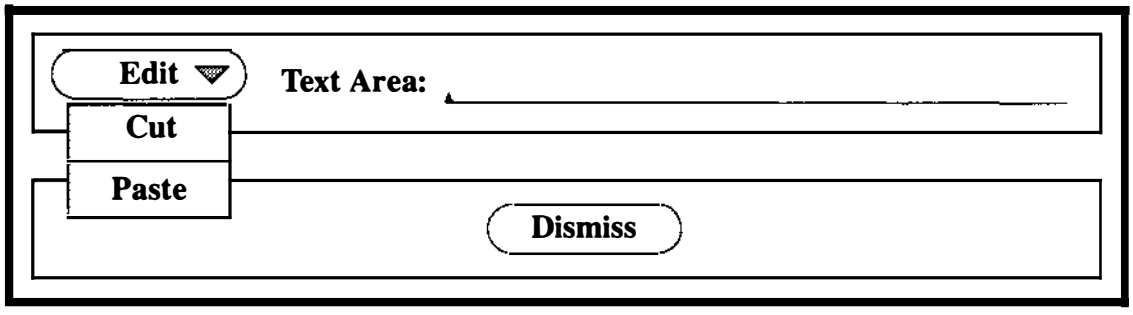


FIGURE 8. An Example Window

user interface of an application, we need to know how various objects in the graphical user interface are organized and how they interact with each other. Building a formal model from this information is helpful in automatic test generation and coverage analysis. Although it is recommended that a formal model of the application under test be built before creating any test data, our methodology and the XTester tool is flexible enough that they work even without such a model. We shall consider two extreme situations as far as modelling is concerned. One extreme is that no effort is spent to build a formal model of the application under test. The other extreme is that maximum effort is spent in modelling the application under test, before creating any test data. In this case, the created model provides each and every detail about the structure and behavior of the graphical user interface of the application under test. Scenarios between these two extremes might include a model which represents the structure of the graphical user interface only, but no information about the behavior.

The first step of the methodology is to make a decision about the model which will be used to represent the application under test and how much effort will be spent in building this model. The model supported by XTester is called User Interface Graph and its specification language is called UIDL (User Interface Description Language). The details about the model and how to build one for an application can be found in [9]. Figure 9 provides such a model for the example window shown in Figure 8.

In Figure 9 the `Root` corresponds to the main window which contains the application. There are two children of root, labeled `Window` and `Menu`. `Window` is the main window of the application and contains all of the elements in the user interface. `Menu` is what appears when the user clicks the button on `Edit`. The Window is composed of two areas, called UpperControlArea, `UpperCA` and LowerControlArea, `LowerCA`. The `UpperCA` contains the text widget, the `Edit` button and the label for the text widget. The `LowerCA` contains the `Dismiss` button. The `Menu` node contains children which are `Cut` and `Paste`. In addition to the parent/child relationships, the User Interface Graph shows which GUI elements are effected by button presses and key presses. For example, both `Cut` and `Paste` can effect the Text widget.

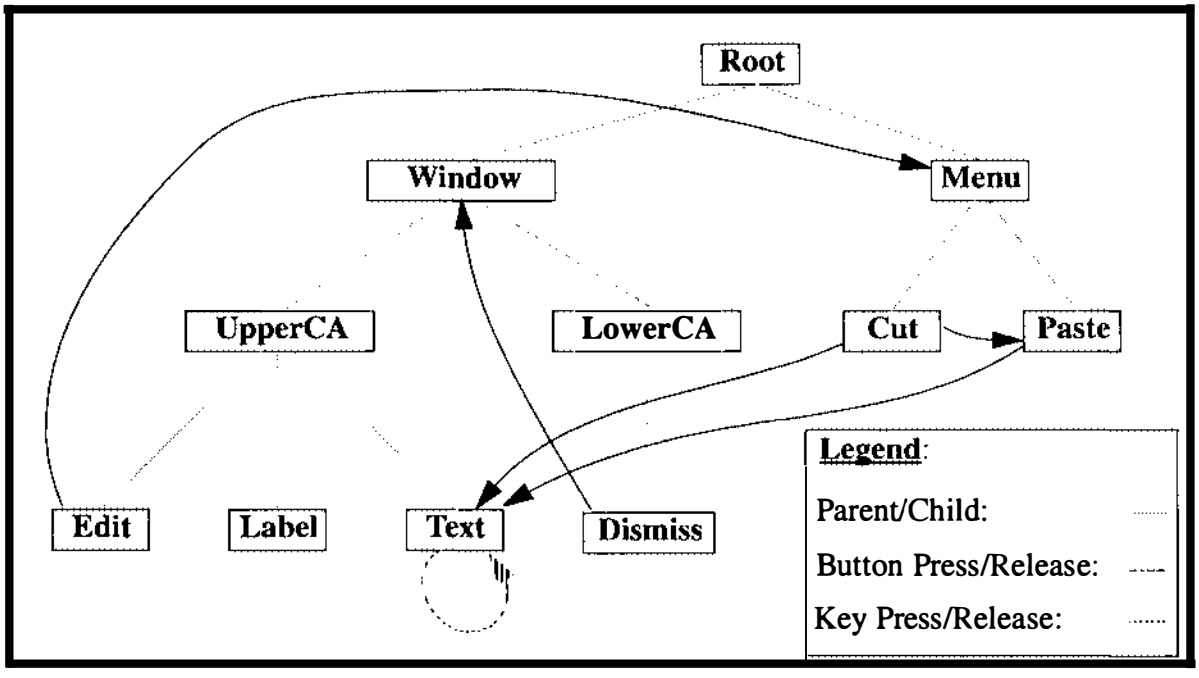


FIGURE 9. User Interface Graph for example window

4.2 Creating the Test Suite

The next step in testing is the creation of a test suite which verifies the behavior of the application being tested. This section specifies some guidelines for creating the test suites for a graphical user interface.

Organize Tests in Different Layers: The objects in a graphical user interface are usually composed of a set of widgets and gadgets, supported by the underlying toolkit. Objects with a common property or related to a particular domain, are grouped together to build higher level objects, (e.g. a set of radio buttons form a radio box). These higher level objects are further grouped together to build another layer of abstraction, (e.g a set of radio boxes and a command button may form a dialog box). These layers correspond to the levels in the object hierarchy of the application. The testing of a graphical user interface should be performed in a layered fashion, starting at the leaves of the object hierarchy and going towards the root. The base layer or the first level of testing should verify the behavior of the individual objects. This is the most important level of testing and most of the user actions will be simulated and captured into scripts at this level. Each of the higher layers will reduce the height of the previous object hierarchy by one, by grouping together leaves with the same parent. The number of testing levels can be reduced by grouping grand-parent or grand-grand-parent, and so on. At each higher level, the interaction and side effects of the objects in the same group are tested. A higher layer will use a combination of the scripts built at the lower layers, in different orders, to verify the effect of user actions on a particular object to the other objects in the same group. The lowest level will always test individual objects while the highest level will group all the top level windows in one group and verify the effects among elements which belong to

different top level windows. At least one level is recommended between these two extremes and that is grouping GUI elements by the top level windows

Organization of Each Layer: Tests at each layer should be organized as a hierarchy of scripts, i.e. files containing commands to simulate user actions and verify results. Each directory in the hierarchy holds scripts which are related to a particular object and its descendents, or a particular group. The individual scripts should be as small as possible and should test one particular feature of an object, or interaction among a particular set of objects in a group.

Organization of a Script: Each script should begin with a clear and precise description of the intended purpose of the script and the state of the application required for its proper execution. A script should be divided into three sections. The first section of the script should build the environment required to run the test properly. It might include commands to simulate user actions on the Application Under Test (AUT) or operating system commands to properly set the environment, i.e. create or rename files and define environment variables. The second section of the script consists of commands required to test the intended feature of the application being tested by the script. This section only contains commands to simulate user actions and verify the results. The third section restores the previous state of the AUT and the operating environment. This section contains commands to restore the state of the AUT and the operating environment to the point when the execution of the script was started. This section might include some commands to simulate user actions, remove some temporarily created files and/or reset environment variables, etc. In this paper, we shall call these sections `Entering`, `Core` and `Leaving` sections, respectively.

Organization of a Directory: The concept of providing information about the script and the state of the environment for its proper execution, and dividing the script in three sections should also be extended to directory level. One possible way of doing this is to provide two additional scripts, an entering and a leaving script, in each directory. These script will not test any particular feature of the application under test but rather will set or reset the stage for proper execution of scripts in current directory and its subdirectories. The entering script is executed on entering the corresponding directory from its parent directory but before executing any script in that directory. It brings the application and the operating environment in a state which is necessary to execute any test in that directory. It also provides information about the directory and the state of the environment on entering the directory. The leaving script is executed just before leaving the corresponding directory to go to its parent directory. It restores the state of the application and its operating environment to a point which existed before we executed the entering script of this directory. All the other scripts except the two mentioned above and any subdirectories belong to the core section of the directory.

Testing of the example window, shown in Figure 8 can be organized from 1 to 4 levels. Each level will correspond to a hierarchy of tests. Let us consider the most conservative approach and perform this testing in 4 levels. Level 1 will test each and every object in the user interface, individually. For example, the `Text` field will be tested for all of its editing capabilities. Similarly, the `Edit` button will be verified for its proper behavior, i.e. if it displays the `Menu`. The next level, level 2, will have three groups corresponding to hierarchies rooted at nodes `UpperCA`, `LowerCA` and `Menu`. The interesting case is the interaction between `Cut` and `Paste`, which will be tested at this level. The third level will have only one group which corresponds to the hierarchy rooted at `Window` node. This will verify if there are any interactions between `UpperCA` and `LowerCA`. The highest

level, level 4 will verify the interactions between `Window` and `Menu` in a single test script, called `Root.scr`.

Figure 10 provides all the four levels for testing the example window, shown in Figure 8, assuming that the root directory of the test suite is called `Example`. The `XTInitDir.scr` and `XTEndDir.scr` scripts in the hierarchy represent entering and leaving scripts, respectively, for the corresponding directory. `Example/start.scr` script contains commands to start the application under test by using all the options which affect the start-up windows. There is only one window in the application, so this script will contain a single command to start the application. `Example/Level-n` hierarchy contains scripts which belong to level n. Scripts in the `Example/Level-1` hierarchy test all the objects in the example window individually. For example `Example/Level-1/Window/Win.scr` tests the main window widget corresponding to the example window. If the main window widget responds to any user actions, then its response is verified in this script. This script also verifies if the Window is created, mapped, unmapped and destroyed properly. `Example/Level-1/Window/UpperCA/Edit.scr` verifies all the editing capabilities of the `Text` widget. Scripts in the `Example/Level-2` hierarchy represents the second level of testing, There are three groups at this level and their corresponding scripts are `UpperCA.scr`, `LowerCA.scr` and `Menu.scr`. The scripts in this hierarchy represent the interaction and side effects among the elements which belong to the same group. For example, `Menu.scr` contains commands to verify that the selection of `Cut` correctly enables the `Paste` selection in the menu but vice versa does not happen. Scripts in the `Example/Level-3` hierarchy represent the third layer of testing and so on. `Example/Level-4/Root.scr` groups the entire application as a single group. This group contains two top level windows, i.e. `Window` and `Menu`, as its members. This script verifies the behavior corresponding to the arcs between these two groups.

Figure 11 expresses our methodology for creating the base layer of a user interface when the application has not been modelled and the testing is performed while steering through the application under test. Figure 12 expresses our methodology for creating a base layer when the application under test has been modelled as a User Interface Graph. Our methodology can be used either to manually create test suites or a tool can be developed to follow the methodology and automatically create the base layer hierarchy. The higher layers are built by using the script at the lower layers.

4.3 Exercising Test Suite and Analyzing Results

Once the test suite has been captured properly and tested for its correctness and coverage, exercising it is fairly simple and should be done after any changes to the graphical user interface. The hierarchical organization of the suite allows us to test the graphical user interface in parts. The layered approach provides us a mechanism to test at various levels of confidence. Exercising of a test suite should be completely automatic and should produce useful error reports and coverage statistics.

XTester provides the capability to exercise scripts automatically. During this automatic replay of scripts, any failures are stored in a log file which can be used later on to determine the cause of the

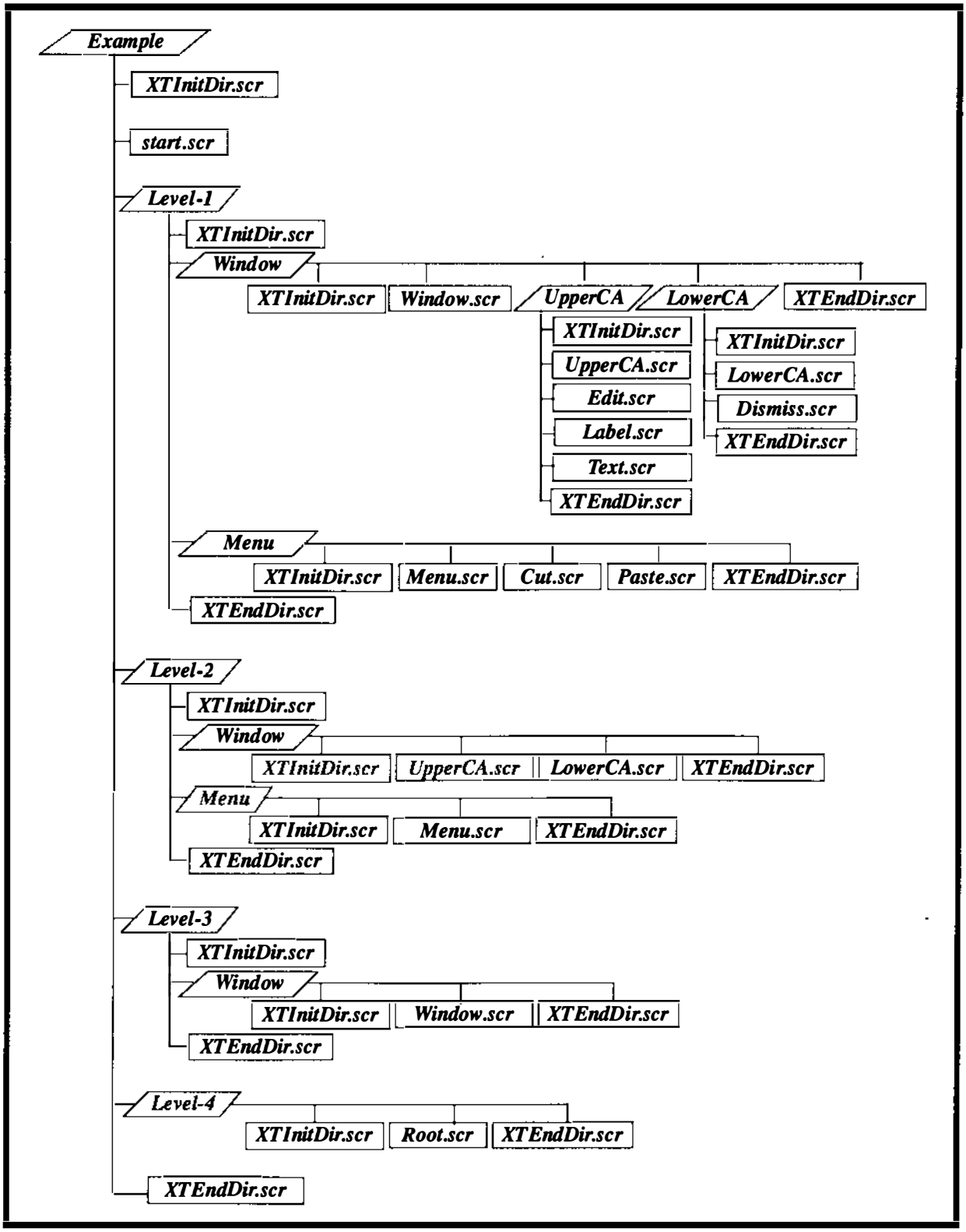


FIGURE 10. Test Suite Directory Hierarchy and Files for the Example

```
Make required changes to the environment;
Initialize a list of window names as empty list;
Start the application under test a number of times using
different options which affect the start-up windows of the
application under test. Whenever an unseen window appears on
the screen, add it to the list of window names, as an untested
window.
While the list is not empty
do
   Get an untested window from the list and mark it tested;
   Create events to get the chosen window on the screen;
   Verify the window image and/or attributes;
   Generate events which affect the window and verify response;
   for each GUI element in the window
   do
      Send expected events to the element and verify response;
      If an unseen object is created as a response
           Add it to the list;
      fi
   done
done
```

FIGURE 11. Strategy for testing without specifications

failure and debugging of the application under test. If a User Interface Graph exists for the application and was used to build the test suite, then XTester collects statistics about user actions on various objects in the user interface. These statistics are stored in a file at the end of exercising the current test suite and can be used to perform coverage analysis.

If the test suite was generated without a formal model of the application under test, then the analysis of the results can be at best guessed by the tester. However if a User Interface Graph for the application exists and was used to build the test suite, then tools can used to automatically determine the coverage provided by the test suite. Such coverage measures might include determining objects which have not been mapped on the screen by the current test suite or any behavior arcs in the User Interface Graph which have not been exercised by the test suite.

5.0 Conclusions

XTester is written in C and provides an OpenLook graphical user interface. The current version runs in a Sun Sparc Station and requires 3MBytes of disk space. We have written test scripts for several applications, including *Framemaker*, a popular UNIX word processing program and

```
Build the User Interface Graph of the application under test;
Build an object list by a pre-order traversal of the User
Interface Graph.
for each element on the list
do
   If the element is a top level window
      Create a new directory and change to the directory;
      Display the element on the screen;
   fi
   if the element accepts any user events
      Create a script for the element;
      for each kind of user event accepted by the element
      do
         Add commands in script to
            Generate the event on the element;
            Verify the effect of that event;
            Undo the effect of the event;
      done
   fi
done
```

FIGURE 12. Pseudo Code for Automatic Test Generator

eXclaim, a popular UNIX spreadsheet program. In both cases we were able to easily capture tests and replay them. Virtually all of the functionality of both programs could be explored using Xtester. There appears to be no performance penalty. We have concluded that XTester is capable of analyzing serious size commercial applications. We are now in the process of determining the effectiveness of the suggested methodology, and attempting to derive criteria to determine the coverage provided by a particular test suite.

References:

[1] XRunner, Mercury Interactive Corp. 3333 Octavious Drive, Santa Clara, Ca. 95054, 1992.

[2] PreVue-X Performance Awareness, 2010 Corporate Ridge, Suite 700, McLean Va. 22102, 1991.

[3] Kepple, Lawrence "A New Paradigm for Cross-Platform Automated GUI Testing", *X Resource, vol. 3, Summer 1992.*

[4] Tillson, Tim and Walicki, Jack "Testing HP Softbench: A Distributed CASE Environment: Lessons Learned", *HP Research Journal, 1991.*

[5] Su, J. and Ritter, P., "Experience in Testing the Motif User Interface", *IEEE Software, March 1991.*

[6] Software TestWorks, Software Research Inc., 625 3rd Street, San Francisco, Ca. 94107, 1991.

[7] OSF/Motif Quality Assurance Test Suite, Open Software Foundation, 11 Cambridge Center, Cambridge, Ma. 02142, 1992.

[8] Robert W. Scheifler and Jim Getty, "The X Window System", *ACM Transactions on Graphics, vol 5, no. 2, April 1986.*

[9] Ellis Horowitz and Zafar Singhera, "XTester Reference Manual", University of Southern California, Los Angeles, CA 90089.

Software Configuration Management Tools: The Next Generation

Steven King
Robert Cohn

Design Technology Laboratory
Tektronix Laboratories
Post Office Box 500, Mail Stop 50-662
Beaverton, OR 97077
(503) 627-2736

sjk@dtl.labs.tek.com
robco@tekig1.pen.tek.com

ABSTRACT

The selection of a commercial tool for Software Configuration Management (SCM) presents many difficult choices. The state-of-the-art commercial SCM tools have change significantly during the last two years and the expectation of the user has been raised. This paper represents an effort to simplify the choices. The authors will first describe the state of the SCM marketplace. We will then present a model which can be used as a guide in choosing the appropriate tool. Finally, we will describe the process that we used in selecting the Tektronix standard SCM tool. The process is designed to focus upon the needs of the users and to select the tool that most closely addresses those needs.

ABOUT THE AUTHORS

Steven King received his BS in Computer and Information Science and Management from the University of Oregon in 1982. He worked at Floating Point Systems and Aptec Computer Systems developing real-time and parallel operating systems, developing software tools and advising on software process improvement. In 1991 he joined the Design Technology Lab to support the Tektronix standard real-time operating system. He currently provides software process improvement guidance to engineering groups throughout Tektronix. He led the effort to select a commercial SCM tool for use in all the divisions of Tektronix.

Robert Cohn received his BS in Computer Engineering from the University of Portland in 1985. He has worked as a software engineer for the Tektronix workstation division developing custom workstation and graphics terminal software and in the Tektronix Logic Analyzer Group performing full-time development and support of configuration management tools. He currently works in the Test & Measurement division, designing user interface software for various instruments. He played a key role in the evaluation and selection of the Tektronix SCM tool.

1. Introduction

In October of 1992 the authors started a project to select a corporate-standard SCM tool for Tektronix. The process used in the selection was a departure from previous efforts. The process was redesigned from the bottom up to focus upon the needs of the software engineers rather than the successful adoption of the tool. The goals were to: 1) include the software engineering community in the process in order to achieve a sense of ownership and 2) to select a tool that works so well that engineers would naturally want to use it.

During the project we found that the SCM marketplace had radically changed during the previous two years. Alex Lobba of CaseWare states that "Attempting to implement CASE without SCM is like having a bicycle wheel with high-tech rim, tire and spokes, but no hub. Modern SCM is the central hub, the foundation that ties together CASE tools in a cohesive software process [Lobb93]." We assert that all software development tools are included in this definition of CASE. Further, we found that modern SCM tools fundamentally changed the way in which a group works. Paul Levine states "While the investment emphasis has previously been on tools that increase individual productivity, the investment focus must shift toward increasing the productivity of the entire development team [Levi93]."

In order to reduce the confusion over the shift in the SCM technology, we will address the state of the marketplace and detail the attributes of current commercial SCM products. We will then discuss our selection methodology, focusing on how we included the software engineers in the process and upon the decision making techniques used in the selection.

2. The SCM Marketplace

Besides the many advances have been made in commercial SCM technology, several new vendors have entered the marketplace. Changes in database technology, user interfaces, CASE tool integration, intra-group coordination and other advances combine to make modern SCM tools a radical departure from the familiar checkin/checkout model of the past. These changes are ultimately good for the discipline of software engineering, yet they make the choice of an SCM tool complex and even more critical to the successful implementation of the technology.

2.1 Advances in commercial SCM technology

The largest advances have come in database technology, tools integration, process support, build management, GUI interfaces and intra-group coordination. The structure of how these features fit within the SCM framework is shown in figure 2-1. As illustrated, modern SCM encompasses much more than simple delta versioning and crude build control. It forms the basis upon which a project is built. Thus, an understanding of the choices available, and the implications for the selection of a particular tool, can lend confidence during an evaluation.

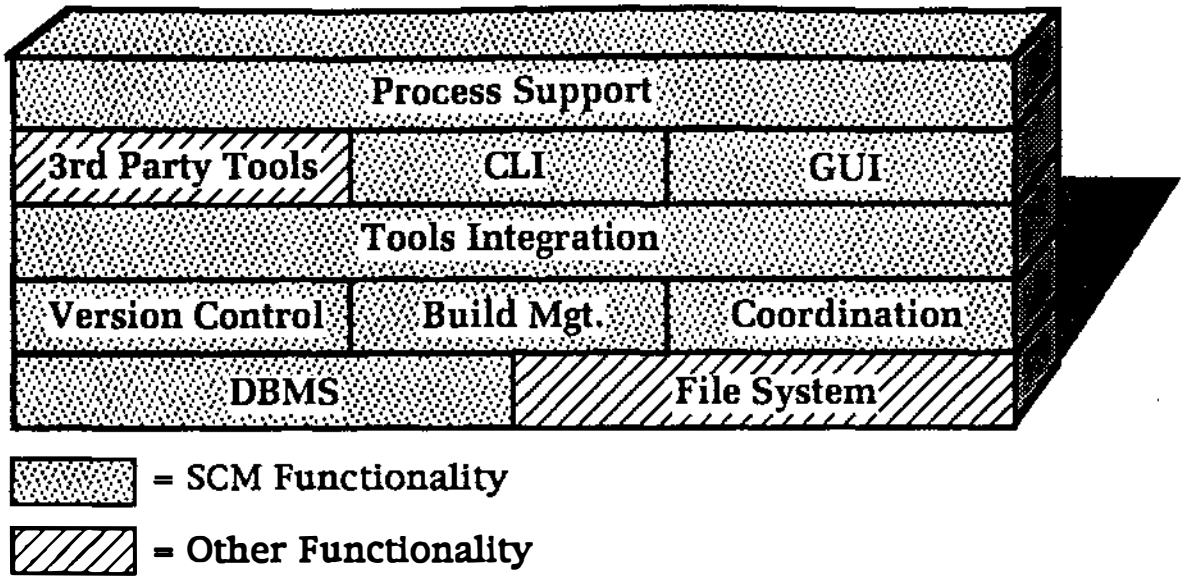


Figure 2-1

2.1.1 Database Technology

SCM tool users need the ability to associate information, or meta-data, with the objects under control of the system. In the past this meta-data took the form of a limited set of values, such as time/date stamps, user id and release identifiers. Due to advances in the database technology used in SCM tools, meta-data can now take any form needed by the user. This allows users to customize the tool to match their development process, instead of molding the process to fit the tool. The DBMS has become a foundation for many SCM tools. Some products, such as Softool's CCC/Manager [Soft92], allow the user to integrate any DBMS into the SCM tool.

Most commercial SCM tools support the association of multiple attributes to an unlimited number of objects. These attributes range from file system oriented date and time stamps to user defined information such as defect IDs and customer names. To manipulate these attributes, the vendors provide query languages. Some provide hooks directly into the DBMS system. These hooks are typically used by the build and audit utilities, and allow the user to build their own tools. In some of the systems, such as CaseWare/CM [Case92], the concept of meta-data permeates every aspect of the product. Examples of this include triggers that are dependent upon attribute values and states, the specification of a view into the hierarchy that is dependent upon selected meta-data and access control that is based upon user identification or the current state of the project.

2.1.2 Tools Integration

Tools integration is currently a hot activity within the CASE marketplace, and SCM tools are no exception. Most of the vendors honor the Tooltalk [Tool91] and HP Softbench protocols. Atherton's Tools Backplane [Athe92] provides a software backplane for inter-tool communication. Many commercial tools provide an API that allows the customer to integrate the tool into their own system, along with integration packages for other CASE tools. Most vendors that currently lack integration with other CASE tools will soon be introducing integration packages.

Some SCM systems are integrated with design tools such as IDE's Software Through Pictures and CADRE's Teamwork, software development environments like Centerline's ObjectCenter and HP's Softbench and documentation systems such as Interleaf and Framemaker. The level at which the integration is supported varies. For example, Teamwork integrates at the class level with some SCM tools. Frame integrates with most SCM tools at the book and document levels. Some SCM vendors are working upon difference tools that support branch-and-merge for non-ASCII (e.g. WYSIWYG) files.

2.1.3 Process Support

SCM is gaining recognition as the foundation upon which a project is built. This shift in thinking has raised the level of emphasis upon the process support of SCM tools. Process support ranges from simple event triggers to scripting languages and process front-ends.

At the most basic level event triggers are necessary for process control. This allows the SCM administrator to associate the execution of a script or tool with any changes made within the code base. These triggers must have access to the DBMS in order to query and change meta-data. Most of the commercial tools pass the meta-data associated with the trigger via environment variables. A script can then use this information without the overhead of a database query. The action performed for a given trigger can and should vary depending upon the state of the project or sub-project. For example, a group may not want development engineers checking in source modules the day before a release! In this case the tool could be used to create a private work area or abort with an error.

The SCM tools that emphasize process do so via GUI front-ends and scripting languages. Tools Backplane supports a GUI that provides the developer with a view into the state of the project first, and then works down to the specific files and tools associated with that state. CaseWare/CM depicts all of the files associated with a particular build target, and displays the working state of each file, directory and build target. Both of these tools support the modification of the process via scripting languages. These scripting languages provide transparent access to meta-data, project state, files, baselines, private workspaces, etc., making the tool easier to customize.

2.1.4 Build Management

The build management capabilities of modern SCM tools far exceed those of the various flavors of make. They include features such as distributed builds, build previews, audit trails, advanced dependency checking, object sharing and build avoidance.

The distribution of builds allows groups to balance the workload across multiple CPUs. The SCM tools that support distribution are of two types: 1) round-robin distribution, where sub-builds are distributed between CPUs in a list and 2) load-balanced distribution, where sub-builds are distributed based upon all the load and processing power of all available CPUs. Some studies have shown that round-robin distribution achieves up to 95 percent of the gain that load-balanced distribution delivers [Case92].

Build preview and audit are very useful features. They allow an engineer, manager or administrator to ask "what if" before starting a build and "what happened" after a build completes. Further, the audit trail is permanently associated with a given object within the database, allowing anyone to discern "what happened" at some point in the future. This is most useful when an incorrect release is produced, yet it "should have worked" since the make log looked correct. The audit trail typically specifies exactly what version of files were pulled into the deliverable object, what version of the tools were used to build the object, and on which CPU they were built. This information can save agonizing debug of a final version of software caused by an incorrect build.

The dependency checking available in modern SCM tools is more advanced than previously available. The checking is typically integrated into the build environment. Some build utilities are so advanced that users often leave dependency rules out of their make files, and instead rely upon the build utility to detect changes and start a build [King93]. This type of dependency checking can, in some tools, automatically invoke the building of other object files or archives, since the dependency used to build the existing version can be checked against the current version(s) of the files from which it was built. With some tools, such as ClearCase [Atri92], if objects or archives that include the exact same components exist in another developers private workspace, it will automatically use that version. This feature allows for the reuse of previous builds, even if they are not in the current workspace. The combination of these and other features help ensure that the software a developer is using is indeed the version they intended, while avoiding unnecessary builds.

2.1.5 GUI Interfaces

The administration of SCM systems has become easier with the addition of GUI interfaces. Since GUI technology is a new addition to SCM tools, the useability of the interfaces varies widely. Some interfaces merely provide alternatives to the command line interface. The better interfaces display information about the selected objects, make good use of object-directed pull down menus to start builds, edit files, view attributes and perform administrative tasks. Most SCM tools provide graphical displays of release hierarchies, private work areas and graphical merge tools, among other useful displays of information.

Most GUIs provide a tangible advantage to release administrators. However, current GUI technology is not an enticement for everyday tool users to migrate from the command line. Most of the vendors are currently working on their next generation of GUIs, and the results should be a step in the right direction for everyday SCM users. We believe that in three to four years the majority of developers will be using the GUI or be driving the SCM tool from an integrated software development environment. In the meantime the command line is the interface of choice for developers.

2.1.6 Intra-group Coordination

One of the most important features of modern SCM tools is intra-group coordination. This feature relies upon the capabilities of the aforementioned features, and varies widely between tools. Most tools provide some form of workspace encapsulation, which allows multiple developers to work in the same code base without interfering with one another. This encapsulation can take the form of the change set model [Feil92], a rule-based view of the system or the transaction paradigm [Feil92]. They provide advanced branch and merge capabilities, allowing controlled integration of changes into a given release. The net effects of this encapsulation are 1) developers do not interfere with one another on a consistent basis, 2) build scripts and/or rules do not change in order to support encapsulation, eliminating a common source of errors and 3) the merging of changes into a given baseline or release of code can be more closely controlled.

Products that feature the change set model, such as TeamNet [Team92] and Aide-De-Camp [SMDS92], assume that changes to a group of files can be viewed as a change set. They often track this change set by a user-defined naming convention, allowing developers to track groups of changes. It is assumed that these changes are then merged back into the baseline of code. This method has the advantage of associating a change or groups of changes to the reason for the change. However, it can limit the flexibility of how a developer obtains a private copy of a file or files. Further, the assumption that a reason exists for a change is not always valid during some stages of product development.

The transaction paradigm [Feil92], as featured in CCC/Manager [Soft92] and PVCS [Inte92], assumes that each change to the system constitutes one transaction. The series of changes can occur over any period of time. They are not associated with a specific reason for the change, but track changes to one or more files as one logical change to the system. Once changes are completed, the transaction is committed to the baseline, at which time everyone has access to the changes. This model has the advantage of simplifying the way in which changes are made, allowing developers to change more files as conditions warrant. It has the weakness of not allowing developers to share any changes until a particular transaction is committed to the baseline.

Tools that use a rule-based system to view of the database, such as CaseWare [Case92] or ClearCase [Atri92], take the perspective that developers need to modify their view of the system on a regular basis. They may need to incorporate the changes made by other developers before the changes are committed to a baseline, for instance. By specifying which baselines, directories and files will constitute a working view of the system, the developer is provided with great flexibility. This flexibility is a double-edged sword, since developers could accidentally incorporate the work-in-progress of others, bypassing the intent of private workspace features.

While modern SCM tools provide many different methods of intra-group coordination, the process always comes down to the need to communicate with group members in order to avoid unpleasant surprises. The degree to which an SCM system prevents these surprises varies between products and work group styles.

2.2 The SCM Tool Model

Given the number of SCM tools on the market, it can be difficult to discern the differences. Following is a simple model that describes the various segments within the SCM marketplace. Figure 2-2 depicts the various segments and identifies where each SCM tool fits within the model.

Following is a description of each segment within the model.

2.2.1 The Checkin/Checkout Model

The checkin/checkout model is best demonstrated by RCS and SCCS. Tools in this category provide revision control of text files, crude control of releases, limited or no branch and merge capability, achieve intra-group coordination through the exclusive locking of files and provide no database capability. For very small projects that only require that a delta history of changes be kept, this class of tool might be considered sufficient. Considering the fact that the first inclination of developers is to extend the capability of these tools, we argue that these tools no longer adequately meet the long-term needs of software engineers.

2.2.2 The Enhanced RCS Model

The enhanced RCS model includes the freeware products Shape and CVS, and the commercial RCS derivative Tower Concepts' Razor. This class of tool provides revision control of many file formats, extended control of baselines, branches and releases, some branch and merge capability, intragroup coordination through the use of links, environment variables and make file magic, and provide some process control capability. For one to five person projects requiring little in the way of database support or integration with other tools, this class of tool could fit the bill. If the developers require a tool that is easily customizable (easy is defined as not modifying the source code), then these tools are probably insufficient.

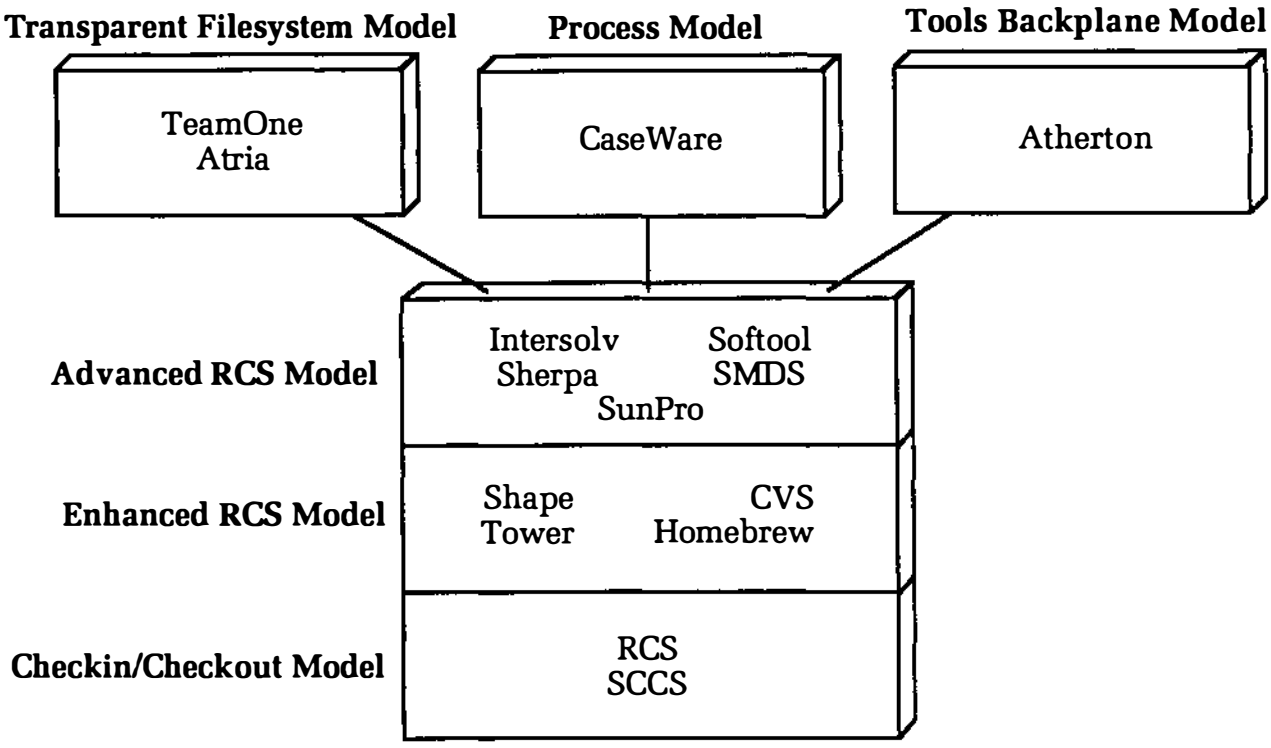


Figure 2-2

2.2.3 The Advanced RCS Model

The commercial products CCC, Intersolv, Aide-De-Camp, SunPro and Sherpa define the advanced RCS model. Conceptually they are all descendants of RCS. These tools feature advanced control of baselines, branches and releases, excellent branch and merge capability, semi-transparent intragroup coordination, easily-configured process control capability, full-flavored DBMS systems, GUI front-ends and integration with other CASE tools. For companies that have multiple teams, each comprised of between five and ten engineers, this class of tool is a definite option. If there exists a strong requirement for intragroup coordination, or build management/auditing then these tools have drawbacks in terms of transparency and ease-of-use.

This class of SCM tool, while it has a definite place in the market, has taken the RCS paradigm about as far as it can go. The next three models radically break the mold of traditional SCM in order to address the problems of process control, build management and intragroup coordination.

2.2.4 The Transparent Filesystem Model

The transparent file system implements many SCM operations within the kernel, controlling the view of the system under control through the filesystem interface. This allows all activity within the file system can be monitored and controlled. It provides a type of safety net that cannot be bypassed by the developers. It also allows the SCM system to control private workspaces, the DBMS, process control and integration with CASE tools almost seamlessly, since existing tools are built on top of the file system.

This class of tool essentially eliminates the problems associated with branch-and-merge and private workspaces. Developers can define their view of the system, make changes to their view of the system, build the system using unchanged build specifications, and do so without interference from (or interfering with) the work of others. If the developer is able to build and test a system in a private workspace, they can be certain that the changes will not break the build or fail the test suite once merged into the baseline.

Since the transparent file system allows the build management system access to transactions at the kernel level, it provides the basis for such features as audit trails, build prediction and avoidance and dependency checking based upon meta-data instead of time/date stamps. The build management system then becomes much more than a time stamp checker, it becomes a tool that anticipates how the system is to be built and permanently records how the system was built.

Through the use of this paradigm, Atria and TeamOne have forever changed how developers view the SCM system.

2.2.5 The Process Model

This model assumes that the SCM system exists to enforce a group's product development process. The management of change and control of building the system are derived from this paradigm. The CaseWare philosophy is to manage the way in which change occurs throughout the project life cycle. To support this view, the tool is customized via a scripting language. The language has access to every feature of the control and build management system. Once the tool has been customized, the developer approaches the system through the process GUI. The interface leads the developer through the process, enforces the process, and provides for intragroup coordination.

The downside to this class of tool is that the up-front investment to define a process and customize the tool can be quite high. Organizations with a mature process will appreciate the focus of this tool. Once the process is defined and implemented, a project can reap large rewards through seamless enforcement of a defined process.

2.2.6 The Tools Backplane Model

The focus of Atherton's Tools Backplane is the integration of the development tools. This is performed via the tools communication back-end and the process front-end. Once the tool is customized the environment is seamlessly navigated using a process-based GUI. This class of tool provides all the rich support of the Transparent Filesystem and Process models. Because the SCM system and tools integration sit above the kernel level, the integration of these systems presents a high price for most users. Once the tool is integrated into an environment, this is perhaps the most elegant SCM system available in the marketplace.

2.3 Future Directions

During the next two to three years, we predict the following changes within the SCM marketplace:

- The lines between the Transparent Filesystem, Process and Tool Backplane models will blur. All the products in this category will offer similar solutions.
- The prices will continue to fall in the Advanced RCS category of tools. These tools will become a requirement for even smallest of projects, since they will be affordable.
- SCM will be viewed as the foundation for enforcement of process, change control, and build management. As such, management will stop asking "why should I pay for a tool when RCS is free?" and require that developers define an SCM environment and choose the tool that best supports their needs.

3. The SCM Project

The SCM Project was a joint effort between Test & Measurement Division and the Design Technology Lab. With two people working an average of twenty hours per week, the project delivered an SCM methodology and SCM tool selection in five months. Since some standard tool selection efforts within Tektronix have had a rather adversarial reputation, we made an up-front decision to redesign the process to avoid previous problems. We identified the following problems with some previous efforts:

- Lack of input from all parties involved
- Lack of communication during the evaluation process
- Limited scope of tools evaluation - not enough requirements, not enough vendors included in the process, etc.
- The software development process of some projects was too immature to effectively use the tool
- Lack of demand for the tool - no early adopters waiting to use the new technology
- Defining a solution or tool where the need is unclear or nonexistent
- Tendency to champion a single vendor too early in the process

- Limited "hands on" evaluation of the technology
- Little or no scrutiny applied to the preferred vendor
- Negotiations handled poorly - little or no discount on pricing, lack of support from the vendor, future contingencies not taken into account
- Lack of support and education for the tool

3.1 Requirements Analysis

To gather input from as many software engineers as possible, we communicated with everyone we knew in the engineering community. We posted notices about the project to internal news groups. We communicated with the leaders of various software organizations. We wrote a project plan [King92] that was distributed to over fifty people, and held a public review of the project plan. We felt that some of the requirements were very "blue sky", but it turns out that some tools actually met these requirements. The feedback that we gathered during this phase proved critical during the evaluation of the various tools.

The requirements were grouped into the following categories: integration with other software tools, object type support, DBMS capabilities, change control, configuration management, build control, reports and displays, administration, and overall technology. We listed the detailed requirements for each of these categories in order to compare the product against the list.

3.2 Initial Evaluation

Initially we included ten commercial products and two freeware tools in the evaluation. Figure 3-1 lists these products. For each of these products we gathered marketing literature and user's manuals. An RFI, which listed the detailed requirements for the SCM tool, was sent to each of the commercial vendors. Seven of the commercial vendors visited Tektronix to deliver a technical presentation and demonstration of their product. Using this information we scored each of the vendors against the criteria.

Using the categories previously described, we assigned a score for each product in each category. We used a simple one through ten rating for each category. Initially we weighted the requirements and found that it didn't change the rank order of the evaluation, so we used a simple addition of the scores. Once the matrix was complete, and the scores compiled, it was clear which products most closely met our needs.

Vendor	Product
Atherton	Tools Backplane
Atria	ClearCase
CaseWare, Inc.	CaseWare
Intersolv	PVCS
Prisma, Inc.	CVS Shareware
Sherpa Corp.	DMS
SMDS, Inc.	Aide-de-Camp
Softool	CCC/Manager
SunPro	SPARCworks/TeamWare
TeamOne Systems, Inc.	TeamNet
Tower Concepts	Razor
University of Berlin	Shape Shareware

Figure 3-1

3.3 SCM Methodology Standards Development

We felt that delivering a recommendation for an SCM tool without some guidelines for the use of the tool would be a recipe for failure. Alex Lobba states "Just focusing on the tools and ignoring methods and practices may accelerate a process that is flawed, thus amplifying existing problems and adding new ones [Lobb93]." Most software engineering organizations within Tektronix do not have a written plan for configuration management or guidelines for the use of a configuration management system. As a part of the project deliverables we wrote guidelines for a Configuration Management Plan and Configuration Management Policy which are based upon the IEEE Standards [IEEE87, IEEE90]. These documents were reviewed by everyone on the Project Plan distribution.

3.4 Early Adopters

During the initial phases of the project we identified software groups that would make good pilot projects for the selected tool. We solicited input from these projects, included them in the vendor presentations and communicated with them on a regular basis. These projects form the set of early adopters of the SCM tool. It was felt that engineers would not accept the recommended tool if other groups were not committed to using it. Further, we could not justify the recommendation if nobody was interested in using the technology. We spent a great deal of time selling our ideas, communicating with the engineers and ensuring that the tool would be a success for these projects.

3.5 Hands-on Evaluation

Based upon our scoring system we narrowed the list of potential vendors to three. We brought each product in-house, installed it and executed the vendor's tutorial. We designed a set of experiments to be run against each product using code from an actual product. This allowed us to evaluate the functionality, perform timings for import and builds and measure disk space usage. We also explored the unique features of the product. We then wrote a subjective report which discussed each tool's strengths and weaknesses.

3.6 References

To ensure that the product performed as advertised we interviewed current users of the products. We spoke with four references provided by each vendor. We also posted a questionnaire to the USENET news group comp.software-eng, which proved to be decisive in eliminating one vendor from the evaluation.

3.7 Final Vendor Visits

Each vendor was invited back for a final visit. We asked each vendor to address the weaknesses that we identified in the product or company, disclose financial information, discuss future plans and share their view of Tektronix as a strategic partner. This information was crucial to making our decision.

3.8 Final Scoring

At this point the purchasing group became involved. They requested quotes based upon the first-year purchase of a large volume licenses. The responses to the quotes were ranked against one another. Using the information gathered during the hands-on evaluation, subjective report, references, financial information and final visit we again scored each product. We added the following categories to the scoring: references, support services, vendor (including financials and working relationship) and cost. Once the technical and pricing evaluation were complete, our choice was clear.

3.9 SCM Tool Announcement

The announcement of the tool was designed to accomplish two objectives. The first objective was to inform the engineering community about the state-of-the-art in SCM tools and the process we used to select the tool. This was accomplished by delivering a talk to the engineering community. The second objective was to impart enough understanding of the tool that engineers would want to evaluate the tool for themselves. This was accomplished by having the vendor visit Tektronix to perform a series of technical presentations and live demonstrations of the tool. The demonstrations allowed engineers to ask questions and see the tool in action.

3.10 Support

We are striving to develop "SCM experts" within each development group. This person will act as a local resource to architect the system, write the SCM policies and procedures, administer the tool, answer questions and solve problems. These engineers attend both the User's and Administrator's classes.

Within the Design Technology Lab we have two SCM tool experts, who spend about 20 hours per week supporting the tool. These people act as an interface to the vendor, answer questions from the various development groups and deliver the education. Over time we anticipate that they will each spend 10 hours per week supporting the tool.

3.11 Education

The vendor was paid to deliver the first series of classes. The Design Technology Lab personell then modified the classes to fit the needs of Tektronix. The User's class, which is aimed for the day-to-day user of the tool (e.g. Software Engineer) can be delivered in one day. The Administrator's class, which is aimed at System Administrators and the group SCM experts, takes two days to complete. It is our goal to develop a cadre of instructors outside of DTL, so that new users can learn from the divisional SCM experts.

4. Results

We feel the selection process was a success. There are currently ten groups with over 90 engineers using the tool. Many other groups intend to adopt the tool sometime in 1993. We anticipate that our projections for usage will be exceeded during 1993 and beyond.

By modifying the selection process to focus upon the user and carefully evaluate the various offering, we achieved the following:

- Met the requirements of the engineers.
- Achieved a high degree of buy-in from the engineers by including them as an integral part of the evaluation process.
- Detected many potential problems with the tools during the hands-on evaluation. These included poor user interfaces, ease-of-use issues, actual bugs in the software, and lack of conformance to the requirements.
- Chose a vendor that is willing to work closely with Tektronix.
- Met, and will continue to meet, the support requirements of the users. We have delivered technical presentations, helped groups install the tool and educated engineers on the use of the tool. We are holding on-site classes for users and administrators of the tool.

As with any process, there are lessons learned that should be applied to subsequent efforts. Based upon the challenges encountered, we recommend that the following changes be made to the process in the future:

- Include system administrators earlier in the process. One system administration group redesigned the way in which they install and maintain tools during our selection process. Once we selected the tool and installed an evaluation copy within their environment, we discovered several minor incompatibilities. Once we understood the requirements, the problems were easily remedied. Including the system administrators up-front would have eliminated this misunderstanding.

- Obtain the resources to perform a full-blown installation and distributed trial of the product. We performed a few, limited, two-person experiments. Based upon these trials and testimony from current users of the product, we were satisfied that the tool would fit into our environment. We were taking a calculated risk, however.

- Obtain purchasing support earlier in the process. We did not obtain purchasing support until the technical evaluation was two weeks from completion. The pricing and contractual negotiations slowed our ability to arrange for education and demonstration licenses. We should have started working closely with purchasing at the start of the hands-on evaluation.

5. Conclusion

By understanding the SCM marketplace and the available technologies, software engineering professionals can make an informed choice of SCM tool. Including software engineers in the selection process and thoroughly evaluating the competing technologies will then lead to selection of the tool most closely matched to the organization's requirements.

6. References

[Adam93] Evan Adams, John Treacy, Simplicity in Software Configuration Management, 4th Annual SCM Workshop, IEEE International Conference of Software Engineering, SunPro Inc. Mountain View, CA, 1993.

[Athe92] Software Backplane: Specification Sheet, Atherton, Inc., Sunnyvale, CA, 1992.

[Atri92] ClearCase Concepts Manual, Atria Software, Inc., Natick, MA, 1992.

[Case92] CaseWare/CM User's Guide, CaseWare, Inc., Irvine, CA, 1992.

[Feil92] Peter Feiler, Alan Brown, Susan Dart, Kurt Wallnau, The State of Automated Configuration Management, SEI Technical Review '92, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.

[IEEE87] IEEE Guide to Software Configuration Management, ANSI/IEEE Std 1042-1987.

[IEEE90] IEEE Standard for Software Configuration Management Plans, IEEE Std 828-1990.

[Inte92] Response to Tektronix RFI, Intersolv, Inc., Beaverton, OR, 1992. (Confidential)

[King92] Steven King, Software Configuration Management Tool Project Plan, Software Technology Research Lab, Tektronix, Beaverton, OR, October, 1992. (Available)

[King93] Steven King, Software Configuration Management Tool Project, Notes from SCM tool reference customer conversations, Tektronix, Beaverton, OR, February, 1993. (Confidential)

[Levi93] Paul Levine, Cost of Quality, Workstation News, Atria, Inc., Natick, MA, April, 1993.

[Lobb93] Alex Lobba, Foundation for CASE, Workstation News, CaseWare, Inc., Irvine, CA, April, 1993.

[Soft92] Response to Tektronix RFI, Softool Corp., Manhattan Beach, CA, 1992. (Confidential)

[SMDS92] Response to Tektronix RFI, Software Maintenance & Development Systems, Inc., Concord, MA, 1992. (Confidential)

[Team92] TeamNet Reference Manual, TeamOne Systems, Inc., Sunnyvale, CA, 1992.

[Tool91] Solaris OpenWindows: The Tooltalk Service, SunSoft, Inc. Mountain View, CA, 1991.

Complexity in Interface Modeling: A Comparison of Two Structural Forms

Patrick Loy
Loy Consulting, Inc.
P.O. Box 24648
Baltimore, MD 21214
410-426-1996
76326.3275@compuserve.com

Abstract: Modeling the interface relationships between system and software components is a critical activity of software development. *Directed graphs* and *connection matrices* are two graphical forms used to model such interface information. These two forms differ substantially in terms of structure and the role that it plays in assigning meaning to the graph. Data flow diagrams (DFD's) and N^2 charts are representatives of these two forms, and are in common use among systems analysts.

This paper discusses a research project which compared these two structural forms with respect to ease of learning, ease of interpretation, and ease of creation. Follow-up activities conducted after the study, the results of which tend to corroborate the research findings, are discussed. The implications that the study and follow-up activities have for software engineers who employ analysis methods, or use CASE tools, are presented.

Keywords: Analysis methods and tools, CASE tools, domain modeling, interface. modeling, knowledge acquisition, software methods, system modeling.

Biographical sketch: Patrick Loy runs his own consulting firm, specializing in development culture evaluation, and improvement strategies. He also serves part-time on the faculty of The Johns Hopkins University where he teaches graduate courses in software engineering. His research interests include eliminating ambiguity in user-developer interaction, and promoting cultural change in the development environment.

Mr. Loy is the author of two technical courses offered by Technology Exchange Company (Addison-Wesley). He has a B.S. in mathematics from the University of Oregon, and an M.S. in computer science from The Johns Hopkins University. He is a member of ACM, IEEE, Computer Professionals for Social Responsibility, and Greenpeace.

Complexity in Interface Modeling: A Comparison of Two Structural Forms

Patrick Loy
Loy Consulting, Inc.

Introduction

Developing high quality software usually necessitates modeling the relationships between various system and software components. Consequently, the value of a modeling schema is determined largely by how well it enables component-to-component interface information to be easily represented and understood.

This paper reports on an experiment, and some follow-up "field work," aimed at assessing the impact that two different structures of graphic knowledge representation schemata have on ease of learning, ease of interpretation, and ease of creation. Our study compared data flow diagrams (DFD's) and N^2 charts, both of which are currently in use by software and systems analysts. However, these two techniques are representative of two general classes of graphical schemata, and thus the implications of our findings have significance for all software or system professionals who employ graphical modeling methods.

Two classes of structural forms

For our study we considered two classes of schemata for system and software modeling. One class consists of *directed graphs* (*digraphs* for short) [1], in which the placement on the page of system components (*nodes*) and connection symbols between the components (*edges*) is left to the discretion of the modeler. The content of interfaces, such as the name of a data flow, or the exact service request between objects, can be shown by labelling the connection symbol accordingly. This class contains the vast majority of the graphical representations in use today, including DFD's and most of the object-oriented analysis and design diagramming techniques. Throughout this paper the term *digraph* will be used to refer to this class of schemata.

The other class uses a *connection matrix* to display interface information between components. In these schemata every location in the matrix has a specific meaning regarding the interfaces between components. Consequently, the meaning of any entry on the page is determined explicitly by its location. Although tabular forms of expression have been used extensively in our field for some purposes, such as specifying a process with a decision table, they have not been used much to model interface relationships between components. However, there are some notable exceptions, such as Lockheed Missiles and Space Company, which makes extensive use of N^2 charts, and even has a proprietary CASE tool to support their use. The term *connection matrix* will be used to refer to this class of schemata.

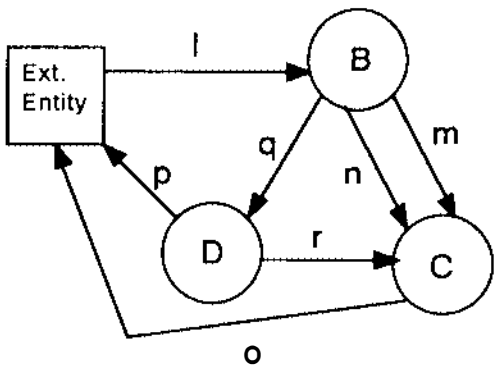
Description of DFD's and N^2 charts

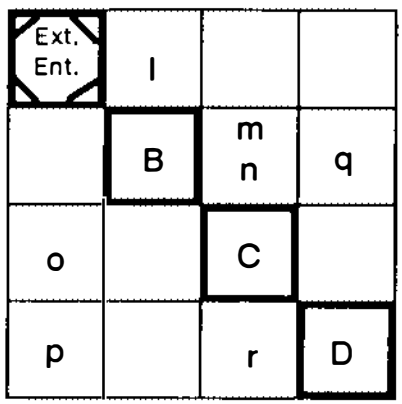
Since DFD's and N^2 charts are undoubtedly the most commonly used representatives of each class, we chose to compare them in our study.

The original concept of DFD's included symbols for data flow, process, data store, and external entity [3,4]. One can think of the latter three symbols as being the *components* on a diagram, with data flow being the connection mechanism between the components. The components form the nodes of the digraph, and the data flows form the edges. Although additional symbols have been added to DFD's over the years, especially the real-time extensions [6], we wanted to minimize the burden on our subjects of learning and interpreting specific symbols - our purpose was to focus on the impact of structural differences. Consequently, we decided to use only the minimum set of symbols necessary to give the schema a dynamic meaning, and therefore did not use any extensions to the original symbol set. Moreover, we felt that having two kinds of internal system components (processes and data stores) would lengthen the learning curve for our subjects, and might detract from our ability to adequately test for structural impact during our limited (two-hour) testing period. Consequently, we used only the symbols for data flow, process, and external entity. In addition, we felt that using only one box for all external entities, and using the abstract name "external entity" for that box, would ease the learning curve of the subjects without sacrificing the integrity of the study.

An N^2 chart [5] is a square matrix where components (processes, external entities, data stores) are placed on the diagonal and the interfaces between those components are put in the other boxes. For our study, we highlighted the diagonal boxes and annotated the external entity box as shown in the figure below. Outputs from a component are always on the horizontal, and inputs to a component are always on the vertical. Thus, if you wanted to know if component B provides any output that is used by component C you would find the box on the matrix where the row containing B and the column containing C intersect. If there were two or more such outputs from B to C they would all be in the same box. Thus, every box in the matrix has a special meaning - it is either on the diagonal or the interface between two components that are on the diagonal. N^2 charts can be partitioned like DFD's with each box on the diagonal potentially becoming a lower level chart.

In the following figure the DFD and N^2 chart have identical meanings.





Objectives of the study

Our project was interdisciplinary in nature, with the author being a computer scientist and the author's collaborator, Dr. Yvonne Stapp, a linguist. We shared a common interest in pursuing the area of ambiguity in knowledge representation and communication, and the role that structural forms play in this area. The author's main interest in the findings of the study was the implications for software engineering, especially the manner in which essential problem-space information is captured, organized, and communicated during requirements analysis. Dr. Stapp was mostly interested in the implications such a study holds for understanding mind/brain processes [2], and is analyzing the results from that perspective for presentation to the cognitive science community.

Characteristics of the subjects

In our study we tested 19 subjects who had no previous experience with either DFD's or N^2 charts, and had varying backgrounds and life experiences. The subjects included five undergraduate students, one graduate student, two physicians, one photographer, one unemployed person, one math teacher, one mathematician, one mechanical engineer, one social worker, one parenting skills teacher, one labor union executive, two receptionists, and one high school student. Eleven were male, and eight were female. All had college degrees except the five undergraduates and the high school student. Seven were non-native English speakers. The median age was 30.6 years, the ages ranging from 14 to 49. None of the the subjects had formal training in computer science, nor had they worked in the computer field. Two testing sessions were held six days apart, the first involving six subjects, and the second, the remaining thirteen.

Teaching Process

At the beginning of the two-hour session we gave the subjects a five-minute orientation to the purpose of the project, and the manner in which the test would be conducted. We explained that they would be asked to perform eight activities, and to respond to some written questions about each of them. We encouraged them to ask questions during the teaching sessions.

During the initial teaching period, which lasted about 30 minutes, the two techniques were taught in the somewhat scaled-down form described above. We took care to give "equal time" to each technique, repeatedly going back and forth between them as we presented increasingly more complex examples. Every schema we presented was followed by its equivalent representation using the other technique.

Our early examples were relevant to their everyday lives (e.g., cooking a meal), with subsequent examples becoming increasingly less familiar and more complex. We focused attention strictly on understanding the mechanics of the two graphical techniques.

Testing process

Following the teaching session we tested the subjects on their ability to interpret DFD's and N^2 charts. During this period, which lasted about 40 minutes, each subject performed four interpretation tests, two each for DFD's and N^2 charts. A time

limit of 5-8 minutes was imposed for completing the answers for each test. After each of the first two tests only, the correct answers were gone over with the group, and the subjects were allowed to ask questions, but they were not allowed to change their answers.

During the interpretation test period the entire subject group was divided into two sub-groups. For the first test one sub-group was given a DFD and the other given an N^2 chart of identical content. In addition, both sub-groups were given sheets containing exactly the same questions, mainly focused on interpreting interface information. For the second test the sub-group that had interpreted a DFD on the first test was given an N^2 chart to interpret, and vice versa. This same pattern continued for the third and fourth tests as well. An example of the interpretation questions used for one of the tests is shown in Attachment 1. (Note that the term *bubble chart* was used for DFD, and *square chart* was used for N^2 chart.)

Following the interpretation tests the subjects were given a short break, after which they were tested on their ability to produce both types of graphs. The production test period lasted about 40 minutes. The two sub-groups were preserved during this period and the tests were administered in exactly the same way as the interpretation tests: each subject alternated between the two techniques, and ultimately created a total of four graphs. For N^2 chart production the subjects were given a blank grid of an 8x8 matrix and instructed to use whatever portion of it they needed. Overall, the subjects had very little difficulty creating either DFD's or N^2 charts. Attachment 2 is an example of a production exercise.

At the end of the entire testing phase we encouraged the subjects to discuss their views about the two techniques before they left.

Findings of the study

In discussing our findings it is important to recognize the limitations of the study. First, we had a fairly small number of subjects, and the interpretation of the data must be seen in that light. Second, our only criterion for subject selection was that they not have had any previous experience with either technique. We did not try to control for any other subject attributes, some of which might well have had a bearing on the results. Consequently, in evaluating the results of our tests we did not try to differentiate based on any sub-group attributes (e.g., sex, academic background, vocation, etc.) Third, the extent to which naive subjects can learn and use such techniques in a two-hour period is necessarily limited. Many subjects commented that their performance might be different in a longer, more in-depth study.

In light of these limitations the results of the study should be viewed as preliminary. However, our confidence in the findings has been bolstered by the results of some follow-up activities, as explained in the next section.

The following are the four findings we deem to be the most significant:

1) Constantly switching back and forth between the two techniques as they were taught did not seem to confuse the subjects at all, and in fact seemed to aid in learning them both. It appears that the two schemata can be taught as a "package"

effectively. Many subjects commented that presenting the two techniques in this kind of "symmetrical" manner greatly assisted them in understanding the two structural forms.

2) By the end of the two-hour test period virtually all subjects had a solid basic understanding of the mechanics of the two techniques, which seemed to indicate that the test activities themselves were efficient learning tools. Even though the interpretation tests became increasingly more difficult, Table 1 shows that the average percentage of correct answers consistently increased (with the exception of DFD test 4, which took a slight dip) and the subjects' perception of ease-of-use moved consistently toward the easy side of the scale.

The practical implication of this finding is that instead of teaching DFD's or N^2 charts in the context of teaching analysis, as is usually done, it might be better to initially focus on the simple mechanics and structure of the methods for a short time, using rigorous interpretation and production exercises.

Interpretation Test Composite Results

Test #	N^2 per cent correct	N^2 ease-of-use*	n=	DFD per cent correct	DFD ease-of-use*	n=
1	60	2.27	9	72	2.05	10
2	66	2.4	10	76	1.94	9
3	70	2.0[1]	9	89	1.85	10
4	76	2.0[2]	10	85	1.5[3]	9

Notes:

* ease-of-use refers to the subjects' rating on a 1-4 scale from *very easy* to *very difficult*
[1] three out of nine subjects did not answer
[2] one out of ten subjects did not answer
[3] three out of nine subjects did not answer

Table 1

3) DFD's were somewhat easier to learn, and to interpret, than N^2 charts. Note in Table 1 that the scores on all four tests were better for DFD's than for N^2 charts. In discussions with subjects after the test they indicated that this was because the flow

of data is explicitly shown on DFD's (source, sink, direction), whereas N^2 charts require remembering structural rules to gain interface information. However, some subjects felt that for more complex systems N^2 charts would be easier to interpret because the "free-form" nature of DFD's would make it more difficult to locate interface information. Also, many subjects commented that after a bit more exposure to N^2 charts they would probably become as easy to interpret as DFD's.

4) N^2 charts were somewhat easier for the subjects to produce than DFD's. However, this finding does not emerge from the performance data, but rather from preferences that the subjects indicated after completing the tests. Overall, the subjects had very little difficulty creating either kind of schema, and no trends were observed from their produced graphs except that they generally improved a little with each new task. However, there was one notable exception to this pattern. During the second test session, five of the 13 subjects had much difficulty on the final production exercise, three of whom were creating a DFD, and two an N^2 chart. Four of these five were non-native English speakers and complained afterward of fatigue due to the language problem. Prior to that fourth and final test the entire group of 13 had consistently improved on each successive exercise. It is likely that fatigue was indeed the problem, as all of the six subjects in the first test session, none of whom were non-native English speakers, performed the fourth production test perfectly.

Therefore, the author's opinion is that the performance data *per se* gives little insight into the relative ease-of-use issue for schema production. In light of this, it would have been beneficial to get some written feedback to some specific questions on this topic. Unfortunately, the subjects were not asked specifically on their written forms to state a preference. However, some valuable data was gained on a more informal basis. At the end of the second test session, the 13 subjects were encouraged to write comments on their forms, including which technique they preferred for production purposes. Eleven gave written responses, 7 of whom (64%) preferred N^2 charts, with 4 preferring DFD's. Four of the eleven explicitly stated that they preferred N^2 charts for production, but DFD's for interpretation.

No such written feedback was collected from the first test session. However, in informal discussions after the session some of the subjects voiced a strong opinion that they thought N^2 charts were easier to create. No one voiced a strong preference for DFD's for production, but the majority seemed to favor them for interpretation.

The most common reason given by subjects for their preference for producing N^2 charts was that the structural form itself relieves much of the "creative burden" for forming the architecture of the schema. In other words, they didn't have to concern themselves with the layout of the schema itself.

Corroborative follow-up activities

Subsequent to the study the author has had the opportunity to "field-test" some of the findings. During three professional development seminars that he has taught, each of which included presenting structured analysis techniques, the students were taught DFD's and N^2 charts in a manner similar to that described above. A total of 42 students participated in the seminars, all of whom were practicing software professionals. None of the students had had extensive experience with DFD's,

about half of them having had no exposure at all. Four students had some prior exposure to N^2 charts, one of whom had used them extensively on the job.

No written feedback was obtained from the students, but the instructor regularly encouraged verbal feedback. The results of the verbal responses seem to strongly support the findings discussed above.

There seemed to be unanimous consent among the students that teaching the two techniques by going back and forth between them helped to facilitate the learning of the structural rules for both. Many students verbalized this view, and most of the others indicated concurrence. No one disagreed with this assessment. Three of the students who had had a prior course on DFD's said that this approach was a better way to learn the basic DFD structural rules.

After the initial teaching phase the students were asked to perform some transformations from one schema to another. These activities were handled slightly differently among the three seminars. In two of them, the students were asked to first take a DFD and turn it into an N^2 chart. After going over the answer and discussing it they were asked to do the opposite and turn an N^2 chart into a DFD.

In the third seminar, the entire class of 14 was divided into two sub-groups of seven each, with one sub-group initially turning a DFD into an N^2 chart, while the other did the reverse. After discussing the solution each sub-group then performed the opposite task.

In all three seminars, after discussing the final solutions, the students were asked which activity had been easier to do, and why. About 30 of the students either said that turning a DFD into an N^2 chart was easier, or gave some gesture of agreement to that view. The rest seemed to feel that the two tasks were about equally difficult. No one said that it was easier going from an N^2 chart to a DFD. In one seminar the students were timed on the task, and 7 of 8 turned a DFD into an N^2 chart faster.

Upon asking those 30 students why going from a DFD to an N^2 chart was easier, several of them repeated the essence of the third and fourth findings of our study: The first part of the task involved interpretation of the given graph which, at least for a novice, is easier with a DFD because the interfaces between components are explicitly shown with labelled vectors. (It should be pointed out that about six students in this group felt N^2 charts were just as easy to interpret as DFD's.) The second part of the task, producing a graph, is easier with an N^2 chart because the structure of the chart relieves much of the creative burden; that is to say, the creator does not have to spend time figuring out how to arrange the information on the page. No one expressed disagreement with this point.

Finally the students were asked which technique they would prefer if they were doing the initial modeling of a system. About 32 of them either said that they would prefer N^2 charts, or expressed concurrence with that view. Five students (all of whom had some pre-seminar exposure to DFD's) said that they would prefer to use DFD's, and the rest were neutral. The reason given for the majority view was that the modeler could concentrate on capturing information without needing to worry much about the

layout of the schema. One student who had used N^2 charts extensively on the job expressed profound satisfaction with them for just that reason.

Conclusions

Probably the most interesting, and strongest, conclusion to be drawn from our work is that *connection matrix* techniques could play a very useful role in system modeling. Up to now, software developers have largely ignored the potential benefits that might accrue from these techniques. In this respect it is not unlike the manner in which object-oriented methods were largely ignored until quite recently, even though the concepts had been around for over two decades.

There could be many reasons why *connection matrix* techniques have been under-utilized. DFD's preceded the advent of N^2 charts by a few years, and were much more widely publicized. Public seminars that taught DFD construction began to flourish as early as the late 1970's. The emergence of DFD-oriented CASE tools undoubtedly had a major influence, and as object-oriented modeling techniques emerged they followed the *digraph* pattern.

So, what role should *connection matrix* techniques play? The most likely manner in which they might be helpful is in the process of partitioning the system into essential components. Armed with blank grids the analysts might well be able to cast the captured information into a meaningful schema faster and easier than using a *digraph* technique. Then, as the analysis work proceeds, the matrices could easily be transformed into another form, if required. It should be easy for CASE tools to automate this transformation.

One final point. Our work should not be interpreted as having found a replacement for DFD's and other *digraph* schemata. Again, the correlation to object-orientation seems pertinent: just as object-oriented methods are a valuable supplement to the software developer's tool box, but do not necessarily replace existing methods, so *connection matrix* schemata could play a valuable role alongside other forms.

References

1 Aho, Alfred V., et al, *Data Structures and Algorithms*, Addison-Wesley, 1983.

2 Anderson, John, *The Architecture of Cognition,* Harvard University Press, Cambridge, 1983.

3 DeMarco,Tom, *Structured Analysis and System Specification,* Yourdon Press, New York, 1978.

4 Gane, Chris, and Trish Sarson, *Structured Systems Analysis: tools and techniques,* Prentice-Hall, Englewood Cliffs, N.J., 1979.

5 Lano, Robert, *A Technique for Software and Systems Design,* North-Holland Press, New York, 1979.

6 Ward, Paul and Stephen Mellor, *Structured Development for Real-Time Systems,* Yourdon Press, New York, 1986.

Attachment 1

Interpretation questions - Order Processing

Name_____Time_____Time_____

1) How many functions have two or more inputs? _____

2) How many functions have two or more outputs?_____

3) What flow(s) does the function "Process Orders" use to generate its output(s)?

_____

4) Which function uses "invoice" to generate its output(s)?

_____

5) Which function has the most flows attached to it? _____

6) Which function generates "shipped-orders?"_____

7) How many functions have more outputs than inputs? _____

8) How many functions have more inputs than outputs? _____

Indicate the time at this point

Task performed: interpretation of (circle one)

bubble chart

square chart

On an easy-to-difficult scale please rate the task you just performed (circle one)

1	2	3	4
very easy	moderately easy	moderately difficult	very difficult

Attachment 2

Square chart production - Payment System

Student name_____time_____time_____

INSTRUCTIONS

Draw a square chart according to the description found below. To aid in clarity, the description follows these conventions: External Entity is spelled with all caps, flows are underlined, and functions are italicized. On your chart you need not follow the same conventions, and you may abbreviate flow and function names if you wish as long as the meaning remains clear. Feel free to erase or start over at any time.

Description

EXTERNAL ENTITY produces the following flows:
credited-payment and mail-payment go to *Record Payment;*
cc-statement goes to *Accept Funds.*

Record Payment produces the following flows:
payment goes to *Accept Funds;*
commission-note goes to *Pay Commission;*
invoice-data goes to *Dun Deadbeats.*

Accept Funds produces the following flows:
bulk-claim and deposit go to EXTERNAL ENTITY.

Pay Commission produces the following flow:
commission goes to EXTERNAL ENTITY.

Dun Deadbeats produces the following flows:
delinquent-invoice goes to EXTERNAL ENTITY.

When you are finished constructing the chart please indicate the time

On an easy-to-difficult scale please rate the task you just performed (circle one)

very easy	moderately easy	moderately difficult	very difficult

A

AACHEN UNIVERSITY OF TECHNOLOGY, 201
Abstraction, for reuse, 202
ACTIVE VOICE, 4
Adequacy of tests, 111
Agrawal, Kaushal, 154
AIDE-DE-CAMP, 418
APPLIED BUSINESS TECHNOLOGY, 343
AQAP-1 quality standard, 274
Assertion failures, 179
Assessments, 13–38, 90
  Capability Maturity Model, 39, 242
  Following up after, 19
  Of risks, 66
ATLANTIC SYSTEMS GUILD, 1
ATRIA, 419
Audits
  Following up after, 19
  Of quality, 13–38
  Self-auditing for ISO 9000-3, 273–291
Automation of testing, 119, 129, 361–373
Axiomatic testing, 111

B

Bank services, 212
Biffl, Stefan, 212, 317
Bitmap comparisons, 394
Black box testing, 155
BOEING COMMERCIAL AIRPLANE GROUP, 292
Börstler, Jürgen, 201
Brewer, Robert S., 301
BS 5750 quality standard, 274
BUSTER TEST MANAGER, 135

C

C Programming language, 157, 393
  Limitations of, 375
C++ Programming language, 223
Call graphs, for testing, 100
Capability Maturity Model, 36, 74, 297
  Advancing levels, 240–256
Capture and replay, 393
  Limitations of, 362
Career growth, 21
CASCADE SOLUTIONS, 354
CASE Tools, 263
CASEWARE, 419
Causal analysis, 75, 193, 195, 263
CCC, 418
Champions, 20, 85
Change
  Effective, 172
  Environmental, 97
  Expectations for improvement, 83
  Following up, 75, 81
  Inevitability of, 238
  Managing, 354–360
  Rate of, 59
  Too much, 88
  Wave model, 87
Checksums for bitmaps, 394
Chilenski, John Joseph, 292
CITE, 361
Classification of software, for reuse, 201–211
Cleanroom development, 157, 296
CLEMSON UNIVERSITY, 109
COBOL, 214
COCOMO, 126
Cohn, Robert, 411
Collaborative work environments, 50, 307
Commitment, 87
Communication, 49, 60, 90
  About reuse, 214
  Between groups, 79, 416
  For user interface consistency, 190
  With top management, 218
comp.software-eng, 423
Competent Programmer Hypothesis, 98
Complexity
  Mutation analysis and, 105
  Reducing, 355
Compromise, 199
Concurrent engineering, 60
Configuration management, 21, 250, 304, 356, 411-426
Connection matrices, 427–435
Consistency of user interfaces, 185–192
Continuous improvement, 28, 84–95, 299
Continuous processes, 64, 65
CONVEX COMPUTER CORP., 361
Cook, Curtis, 257
Cooperating tools, 208
Costs
  Constraints on, 213
  Labor-intensive reviews, 302
  Low-budget testing, 124
  Meeting objectives of, 61
  Of defects, 77
  Of maintenance, 187
  Of non-conformance to standards, 194
  Of testing, 124, 136
  Of tools, 44
Cousin, Larry, 240, 374
Coverage of test cases, 109
Creativity, 197
  Graphical User Interfaces, 385
CREDENCE SYSTEMS CORP., 74, 193
Crosby, Philip, 75
CSRS, 301–316
Customer involvement, 162
CVS, 417

D

Data entry systems, 355
Data flow diagrams, 427–435
Database, relational, 135
Dataflow testing, 96–108
Debugging, 294
Decision making, 78
  Consensus, 63
Defects
  Attitudes for preventing, 23
  Costs of, 77
  Intermittent, 129
  Metrics for, 21
  Prevention of, 76, 254, 344–353
  Types of, 77
DeMarco, Tom, 1
Design, 91
  For testing, 129
Design documentation, 334
DETROIT MERCY, UNIVERSITY OF, 273
Development for reuse, 202
Directed graphs, 427–435
Documentation
  Assessment guidelines, 15
  Design, 330
  For processes, 27
  Improved by prototyping, 388
  Lack of, 97
  Maturity Model, 257–271

Documentation (*continued*)
Of processes, 250
Templates for, 46, 90
DOCUMENTUM, INC., 234
Donahue, Jim, 234
Duncan, I. M. M., 96
Durham, University of, 96
Dyer, Douglas M., 109

E

Eclectic CMM Progression, 243
Editor, syntax directed, 207
Effort, tracking of, 331
EGRET, 303
Electronic mail, 190
See also, Paperless reviews
EMACS, 303
Embedded systems, 154–170, 272
Encapsulation, for reuse, 202
Errors, *See* Defects
Examples
Of work products, 79
User interfaces, 191
Expectations for improvement, 82
Experts of configuration management, 424

F

Failures, run-time, *See* Defects
Faults, *See* Defects
FAX machines, 5
Ferrell, Ian, 171
Finite State Machines (FSMs), *See* State machines
FOCS, 201–211
Follow up
After quality audits, 19
Of changes, 75, 81
Fonorow, Owen Richard, 135
Food and Drug Administration (FDA), 272
Formal methods, 15, 91, 262, 318
Formal technical review (FTR), 301–316
FORTH, 124
Foster, Lynne, 13

G

Gluch, David P., 58
Goals
Estimates separate from, 3
Goals (*continued*)
Measurable, 253
Objectives and, 5
Quality, 58
Setting of, 75
Success as a, 2
Goals/questions/metrics, 319
GRAIL, 96
Graphical User Interfaces (GUIs), 185–192
Configuration management with, 414, 419
Rapid prototyping of, 374–390
Testing of, 391–410
Grechenig, Thomas, 212, 317
Group meetings, 190

H

HALIBURTON ENERGY SERVICES, 185
Harrison, Roger, 374
HAWAII, UNIVERSITY OF, 301
Hazard rates, 175
Hierarchies
Of tests, 118, 144
Of user interface components, 401
High-volume production, 123
Higuera, Ronald P., 58
Hoffman, Daniel, 223
Hoffman, Douglas, 293
Horowitz, Ellis, 391
Human factors
Graphical User Interfaces, 382
HyperCard, 377
Hypertext, 208, 303

I

IBM, 154
Improvement, 286
Continuous, 28, 84–95
Expectations for, 83
Of processes, 74–83
Of quality, 240–256, 277
Incremental development, 302
indent (UNIX command), 197
INDIANA UNIVERSITY, 48
Inspections, 302
Interface Modeling, 427–435
INTERSOLV, 418
Interviews
For assessing risks, 67
For capability assessment, 277
Invariants, use in testing, 113
Inventory, software as, 349
IPSEN, 208
ISO 9000, 273–291, 298
ISO 9000-3, 46
ISO 9001, 36

J

Johnson, Philip M., 301
Joos, Becky, 84
Jovanovic, Vladan, 273

K

Keiretsu, 41
King, Karen S., 74
King, Steven, 411
KPMG PEAT MARWICK, 328

L

Large organizations, 71, 86
Large systems, testing of, 96–108
LaTeX, 306
Leaders, of teams, 6
Leveson, Nancy G., 10
Life-critical systems, 123
Life cycles, 20, 90, 172, 296
Lister, Timothy, 1
Loy, Patrick, 427

M

Main, F. Beachley, 185
Mainframes, shift away from, 213
Maintenance
Improved by prototyping, 388
Reuse and maintainability, 218
Malcolm Baldridge National Quality Award, 15
Management
Commitment to quality, 23, 82, 88, 188, 213, 273–291
Commitment to UI consistency, 191
Communication with, 218
Involvement, 70, 196
Metrics for, 328–342
Of risks, 27, 58
Support of, 246, 389
Manufacturing, quality techniques, 345
Marketing research, 173

Markov chains, 154–170
Maturity, 297
  Capability/Process Model, *See* Capability Maturity Model
  Documentation, 257–271
Maybee, Joe, 122
McGregor, John D., 109
Mean time to failure, 171–184
Medical devices, 272
METACARD, 377
Metric engineering, 319
Metric Premise, 2
Metrics, 76, 82, 164, 253, 295, 309, 328–342, 343
  Code reviews and, 317–327
  Comprehensive program, 277
  For defects, 21
  For documentation, 263
  For results of changes, 81
  Of reliability, 171–184
  Optimizing success measures, 2
  Quantitative evaluation, 321
  Recording, 358
  Review, 303
  Similarity, for reuse, 206
  Time related, 311
METROPOLIS, 356
MICROSOFT CORP., 171, 344
MIL-Q-9858 quality standard, 274
MIS departments, 44
Morale, 5, 23, 85
MOTHER, 122–134
MOTOROLA, 13, 84
Mutation Analysis, 96–108

N

N^2 charts, 427–435
NATIONAL COMPUTING CENTRE, 42
NCR CORPORATION, 135
Networks
  Configuration management in, 414
  Newsgroups, 190, 423
NOBLITT & RUELAND, 272

O

Object oriented technology, 109, 213, 234–239
Objectives, *See also*, Goals
  Defining, 5
On-line processes, 303
Operational profiles, 173
OREGON STATE UNIVERSITY, 257
Organization size, 86

P

Paperless reviews, 303
Pareto Principle, 351
Participation
  In quality improvement, 23
  In risk management, 70
Peer groups, 68
Peer reviews, 250
People
  Career growth, 21
  Optimizing measures of success, 2
  Performance reviews, 23
  Relations with QA group, 295
  Tension between, 5
Peopleware, 1
Perception
  Of risks, 63
  Of system interfaces, 427–435
Personal Computers
  Turn-key systems, 5
Pilot projects, 90, 250
  For configuration management, 422
Planning, projects, 27, 249
Poka yoke (mistake proof), 344–353
Portability, 97
  Prototyping and, 379
PORTLAND STATE UNIVERSITY, 122
Post project reviews, 195
Predicates, testing of 113
Predicting behavior, 295
Prevention
  For risk management, 64
  Of defects, 23, 254, 344–353
Processes
  Consistent, 194
  Documentation for, 27
  Improvement of, 74–83
Productivity, 127
  Enhancing, 40
Project management, 27, 249, 330

Q

Quality, 2, 274
  Goals for, 58
  Improving, 240–256
Quality (*continued*)
  Of documentation, 258
  Wave model for changing, 84–95
Quality Assurance (QA) group, 86, 250, 293–300
  Relationships with, 295
Quality System Review, 13–38
Quantitative evaluation, 321
Questionnaires
  Documentation quality and, 260
  For assessing risks, 67
  For assessments, 16, 27–32, 277
Quick tests, 348

R

Random testing, *See* Testing, Statistical
Rapid prototyping, 374–390
Rapid testing, 131
RAZOR, 417
Reactive vs. proactive QA group, 294
Real-time systems, 123, 154–170, 272
Record and playback, *See* Capture and replay
Regression testing, 100, 392
Reliability, 164, 171–184
REPARTEE, 5
Repeatability of tests, 338
REPLAY, 5
REPLAY PLUS, 5
Requirements, 27, 91, 166
  Analysis for configuration management, 421
  Defects in, 77
  Embedding tests in, 124
  Lack of, 136
  Specifications for, 260
Resource center, for software, 39
Reuse, 191, 212–222
  Classification to support, 201–211
  Graphical User Interfaces, 385
  Of tests, 140
Reviews, 91, 250
  Of code, 317–327
  Of personal performance, 23
  Paperless, 303
  Post project, 195

REVISION CONTROL SYSTEM (RCS), 136, 417
Risks
  Management of, 3, 27, 41, 58–73
  Predicting, 216
Robson, D. J., 96
Rossman, Ilona, 84
Rubenacker, Dennis, 272
Rule-based systems
  For configuration management, 417
  For testing, 364

S

Schedules, 1
  Slipping of, 5
Sense-Diagnose-Improve, 273
SHAPE, 417
SHERPA, 418
Shingo, Shigeo, 346
Shoemaker, Dan, 273
Siegel, Matt, 171
Singhera, Zafar, 391
Small organizations, 44
  Managing change in, 354–360
Small projects, 8
SOFTWARE ENGINEERING INSTITUTE (SEI), 58, 214
Software Engineering Process Group (SEPG), 74–83, 86, 250
SOFTWARE PRODUCTIVITY CENTRE, 39
SOFTWARE QUALITY METHODS, 293
SOFTWARE TECHNOLOGY, INC., 109
Source code control, 304
SOURCE CODE CONTROL SYSTEM (SCCS), 136, 417
SOUTHERN CALIFORNIA, UNIVERSITY OF, 391
Spencer, Nancy, 343
Spiral model, 243
Standards, 82, 250
  Coding, 193–200
  Configuration management and, 411-426
  Documentation, 46
  For reuse, 218
  User interface, 185–192
State machines, 110, 155
  For requirements, 124
Statistical Quality Control, 194, 253
Statistical testing, 127
Steam engines, 10--12
Stochastic processes, 154–170
Stoiantschewsky, Spass, 193
Strange, Michael, 328
Strategies
  For test coverage, 114
Stress testing, 349
Strigel, Wolfgang B., 39
Style guides, 185
Subcontractors, 21, 28, 66, 249
Sullivan, Sarah L., 48
SUMIT, 161
SUNPRO, 418
SUPERCARD, 377

T

T (test script language), 393
TEAMONE, 419
Teams, 85, 60
  Competition between, 89
  Core, 4–8
  Leaders of, 6
  User interface, 189
Technology transfer, 41
Teisher, James, 193
TEKTRONIX, INC., 122, 411
Teleconferencing, 190
Telephone systems, 5
Templates
  For code, 127
  For documentation, 46, 90
  For work products, 79
Test plans, 262
Testgraphs, 227
Testing, 294, 330, 336
  Adequacy of, 111, 164
  Aided by prototyping, 388
  Automated, 119, 129, 171–184, 223–233, 361–373
  Black box, 112, 155
  Class libraries, 223–233
  Completeness of, 164
  Consistency of, 131
  Costs of, 124, 136
  Coverage, 99, 109–121, 128, 337, 348
  Critical regions, 99
  Databases for, 135, 367
  Graphical User Interfaces, 391–410
  Large numbers of, 137
  Low budget, 124
  Multiple environments, 139
Testing (*continued*)
  Mutation analysis, 96–108
  Rapid, 131, 348
  Regression, 100, 392
  Selecting test cases, 109–121
  Statistical, 127, 154–170, 173
  Stopping criteria, 106
  Stress testing, 349
  Time for, 349
  Usability, 384
3M HEALTH INFORMATION SYSTEMS, 240, 374
Tierney, James, 344
Timestamp metrics, 311
Tjahjono, Danu, 301
Tools, 356
  Collaborative work environments, 301–316
  Computer zoo, 43–44
  Cooperating, 208
  Documentation, 262
  For rapid prototyping, 374–390
  For reuse, 210
  For testing, 225, 337, 364
  Group for, 21
  Lack of, 318
TOOLS BACKPLANE, 420
Total Quality Management, 13, 260
  See also, ISO 9000
Traceability, of requirements, 124
Training, 91, 214, 263, 337
  For configuration management, 424
  For prototyping tools, 377
  For software processes, 79
  For standards, 194
  For user interfaces, 189
  In CMM, 248
  In statistical testing, 156
  Software engineering, 42
Transparent filesystems, 419
Trends
  Observing, 175

U

UIDL, 401
UNIX, 303
Usability, 187
  Enhanced, from prototyping, 388
  Testing of, 384
Usage profiles, 173, 263

Usage profiles (*continued*)
Comparing tests and users, 177
User-centered design, 374–390
User interface
Consistency of, 355
Of tests, 132

V

Vernum, Laura, 4
VICTORIA, UNIVERSITY OF, 223
VIENNA, TECHNICAL UNIVERSITY OF, 212, 317
Visconti, Marcello, 257
VISUAL APPBUILDER, 377
VISUAL BASIC, 377
Vogel, Peter A., 361
Voice mail systems, 5

W

Wan, Dadong, 301
WASHINGTON, UNIVERSITY OF, 10
Wave model of quality, 84–95
WEASEL, 178
White, Donald H., Jr., 354
Whittaker, James M., 154

X

X-Windows, 303, 392
XTESTER, 391

Z

Zoo, computer, 43–44

1993 PROCEEDINGS ORDER FORM

PACIFIC NORTHWEST QUALITY SOFTWARE CONFERENCE

To order a copy of the 1993 proceedings, please send a check in the amount of $35.00 to:

PNSQC
PO Box 970
Beaverton, OR 97075

Name _____

Affiliate _____

Mailing Address _____

City _____

State _____

Zip _____

Daytime Phone _____