# THIRTEENTH ANNUAL

# PACIFIC NORTHWEST

# SOFTWARE QUALITY CONFERENCE

## September 27 - 29, 1995

## Oregon Convention Center
## Portland, Oregon

# TABLE OF CONTENTS

**Preface** . . . . . vi

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

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

**Presenters** . . . . . ix

**KEYNOTE - September 28**
*"Prospects for an Engineering Discipline of Software"* . . . . . 1
Mary Shaw, Carnegie Mellon University

**KEYNOTE - September 29**
*"A Cleanroom for Your Mother"* . . . . . 11
Dr. Charles B. Engle, Jr., Florida Institute of Technology

**MANAGEMENT TRACK - September 28**

*"Software Process Assessments Around the World"* . . . . . 21
Gordon Wright, American Management Systems, Inc.

*"Finding, Qualifying, and Managing Software Subcontractors within a Large Engineering Project"* . . . . . 35
Jim Nielsen, Motorola

*"Assessment in Small Software Companies"* . . . . . 45
Woldgang B. Strigel, Software Productivity Centre

*"Developing a Useful Framework for Software Metrics"* . . . . . 57
Barbara Zimmer, Hewlett-Packard Company

*"Developing a Low-Defect Software Product in a Research Organization"* . . . . . **68**
Jeffrey M. Bell, Pacific Software Research Center
Laura McKinney, Pacific Software Research Center

*"A Non-Technical Look at the Evolution of Testing at Visa"* . . . . . **78**
Anthony C. Alosi, Visa International

**PROCESS/ENGINEERING TRACK - September 28**

*"A Process Model for Managing Product Support Risk and Design Quality"* . . . . . **89**
Don Moreaux, Hewlett-Packard Company

*"The Quality Improvement Core Team:*
*A Grass-Roots Approach to Process Improvement"* . . . . . **105**
Leslie A. Dent, Synopsys, Inc.

*"Learning from Our Mistakes"* . . . . . **115**
David N. Card, Software Productivity Solutions

*"Cleanroom Software Engineering and 'Old Code'—Overcoming Process Improvement Barriers"* . **126**
Michael Deck, Cleanroom Software Engineering, Inc.

*"The Cleanroom Process Model: Caution Advised"* . . . . . **148**
Boris Beizer, Consultant

**TOOLS/TESTING TRACK - September 28**

*"Workshop Test Infrastructures"* . . . . . **174**
Allyn Polk, SunSoft

*"Automatic Generation of Novice Test Scripts"* . . . . . **176**
David J. Kasik, Boeing Commercial Airplane Group
Harry G. George, Boeing Commercial Airplane Group

*"The Quest for the Right GUI Test Tool"* . . . . . **187**
Phil Allred, Novell, Inc.

*"Writing GUI Specification in C"* . . . . . **214**
Shankar Chakrabarti, Hewlett-Packard Company

*"TRUBAC: Testing Expert Systems with Rule-Base Coverage Measures"* . . . . . **218**
Valerie B. Barr, Polytechnic University

*"ClassBench: A Framework for Automated Class Testing"* . . . . . **238**
Daniel Hoffman, Ph.D., University of Victoria

**MANAGEMENT TRACK - September 29**

*"Jump-Start Your Stalled Improvement Effort"* . . . . . **254**
Hilly Alexander, ADP Dealer Services
Margie Davis, ADP Dealer Services

*"One Small Company's Struggle to Produce Quality Software"* . . . . . **276**
Anton Webber, MSR Development
Michael M. Packard, Ph.D., Stephen F. Austin State University

*"A Software Quality Strategy for Demonstrating Early Return on Investment"* . . . . . **290**
Tim Olson, Cardiac Pacemakers, Inc./Juran Institute

*"Software's Dirty Dozen"* . . . . . **304**
Neal Whitten, The Neal Whitten Group

**ENGINEERING/LEGAL TRACK - September 29**

*"Using Fault Injection to Assess Software Engineering Standards"* . . . . . **318**
Jeffrey Voas, Ph.D., Reliable Software Technologies Corporation
C.C. Michael, Ph.D., Reliable Software Technologies Corporation
Keith W. Miller, Ph.D., Sangmon State University

*"Predicting Fault-Prone Modules: A Case Study"* . . . . . **335**
Taghi Khoshgoftaar, Ph.D., Florida Atlantic University

*"Metrics Measuring Control Flow Complexity in Concurrent Programs"* . . . . . **342**
Shengru Tu, Ph.D., University of New Orleans
Michael E. Mathews, University of New Orleans

*"Analysis Error's Estimation in a Large MIS: Two Empirical Case Studies"* . . . . . **358**
David Gefen, Georgia State University

*"Reliability of Medical Software"* . . . . . **375**
Steven C. Schurr, Attorney at Law

**TOOLS/TESTING TRACK - September 29**

**Software Excellence Award**
*"Developing Quality Processes in a Startup Team at ParcPlace"* . . . . . **397**
Jay Van Sant, ParcPlace Systems
Jenny Greenleaf, ParcPlace Systems

*"Software Testing in a Product Enhancement Team"* . . . . . **409**
Jonathan R. Brandt, Spectra-Physics Scanning Systems, Inc.

*"PARSE: Problem Analysis and Process Management for Software Maintenance"* . . . . . **423**
Robert Bruce Kelsey, Ph.D., Digital Equipment Corporation

*"Selection of a Defect-Tracking System"* . . . . . **434**
Peter Hantos, Ph.D., Xerox Corporation
Raymon T. Li, Xerox Corporation
Sally Robinson, Xerox Corporation

*"Improving Communication Using Groupware"* . . . . . **449**
Brandon Masterson, Cascade Engineering Services
Tim Fujita-Yuhas, Crane Eldec Corporation

*"Reducing Software Schedules with a Process-Based Configuration Management Model"* . . . . . **465**
Karen Parker, Summit Design

**Index** . . . . . **478**

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

# Preface

G.W. Hicks

It is with great honor I welcome you to the Thirteenth Annual Pacific Northwest Software Quality Conference. Throughout its entire existence, PNSQC has been successful because of our attendees and the companies they represent.

As President of PNSQC this year, I have seen our software community grow and evolve. At the same time, I have seen PNSQC grow and evolve.

For the first time, PNSQC held spring workshops in two locations: Bellevue, Washington, and Beaverton, Oregon. The workshops were a great success. This was due to the efforts of the Workshop Committee, chaired by Miguel Ulloa. Great job, Miguel and Committee! The successful completion of a 33-percent increase in workload is outstanding.

This year's program is a fine collection of 23 refereed paper presentations and 9 interesting invited speakers that we hope will provide you with information and insight to use in your personal and professional lives. I would like to thank the Program Committee, co-chaired by Peter Martin and Margie Davis, for the hours of effort it takes to put together such a fine program.

Again in 1995, we have two interesting keynote presentations. On Thursday, Mary Shaw from Carnegie Mellon University will present an address on "Prospects for an Engineering Discipline of Software," which will certainly pique the interests of those in the software engineering community who struggle to apply discipline in their work environments. On Friday, Dr. Charles Engle will enlighten us on Cleanroom principles and practices when he presents "A Cleanroom for Your Mother."

Once again, PNSQC has provided the software community of the Pacific Northwest with events true to its mission to provide low-cost, high-quality educational and networking opportunities.

Working on PNSQC this year has been a privilege. I have worked with a group of fine professionals, and I have benefitted from it. As you search the pages listing the volunteers who together made this year's conference a reality, imagine your name on the list. Working on this conference can be very rewarding professionally and personally, which is why I encourage you to consider working on PNSQC.

Finally, I would like to thank the unsinkable Terri Moore and Pacific Agenda for being the superglue that cements the efforts of many into affordable access to information and events relevant to the software quality community.

## CONFERENCE OFFICERS/COMMITTEE CHAIRS

**G.W. Hicks - President/Chair**
*IMS, Inc.*

**Ian Savage - Secretary**
*CFI ProServices*

**Ray Lischner - Treasurer**
*Tempest Software*

**Margie Davis, Program Co-Chair**
*ADP Dealer Services*

**Peter Martin, Program Co-Chair**
*Taligent, Inc.*

**Miguel Ulloa - Workshops**
*Mentor Graphics*

**James Mater - Exhibits**
*Revision Labs, Inc.*

**Rick Clements - Publicity**
*Tektronix, Inc.*

**Barbara Siefken - Software Excellence Award**
*Tektronix, Inc.*

**Karen King - Birds of a Feather**
*Sequent Computer Systems, Inc.*

## CONFERENCE PLANNING COMMITTEE

**Hilly Alexander**
*ADP Dealer Services*

**Judy Bamberger**
*Sequent Computer Systems, Inc.*

**Susan Bartlett**
*Tektronix, Inc.*

**Erin Davis**
*ADP Dealer Services*

**Margie Davis**
*ADP Dealer Services*

**Dave Dickmann**
*Hewlett-Packard Company*

**Dolores Emery**
*PenMetrics, Inc.*

**Lynne Foster**
*Motorola*

**Cynthia Gens**
*Technical Solutions*

**Mike Green**

**Georgette Hauck**
*CFI ProServices, Inc.*

**Dick Hamlet**
*Portland State University*

**Warren Harrison**
*Portland State University*

**Roxie A. Hecker**
*ADP Dealer Services*

**Dan Hoffman**
*University of Victoria*

**Craig Hondo**
*IMS, Inc.*

**Connie Ishida**
*Sequent Computer Systems, Inc.*

**Mark Johnson**
*Mentor Graphics Corporation*

**Bill Junk**
*University of Idaho*

**Karen King**
*Sequent Computer Systems, Inc.*

**Randolph King**

**Ray Lischner**
*Tempest Software*

**David Loseke**
*Tektronix, Inc.*

**Peter Martin**
*Taligent, Inc.*

**James Mater**
*Revision Labs, Inc.*

**Joe Maybee**
*Portland State University/Tektronix, Inc.*

**Paul McCullough**
*Penergy, Inc.*

**Howard Mercier**
*Intersolv*

**DeAnn Miller**
*ADP Dealer Services*

**Jerry Pitts**
*Credence Systems Corporation*

**Ian Savage**
*CFI ProServices, Inc.*

**Eric Schnellman**
*CDP*

**Bill Sundermeier**
*FLIR Systems, Inc.*

**Jim Teisher**
*Credence Systems Corporation*

**Andrew Tolmach**
*Portland State University*

**Don White**

**Scott A. Whitmire**
*Advanced Systems Research*

**Barbara Zanzig**
*NeoPath, Inc.*

**PROFESSIONAL SUPPORT**

**Linda Cordelia**
*Graphic Design*

**Pacific Agenda, Inc.**
*Conference Management*

## PRESENTERS

Hilly Alexander
ADP Dealer Services
2525 SW First Avenue
Portland, OR 97201-4760

Phil Allred
Director, Engineering Services
Novell, Inc.
MS: B-231
1555 N. Technology Way
Orem, UT 84057

Anthony Alosi
Vice President
Visa International
P.O. Box 8999
San Francisco, CA 94128-8999

Valerie Barr
Department of Computer Science
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11202

Jeff Bell
Oregon Graduate Institute
P.O. Box 91000
Portland, OR 97291-1000

Jonathan Brandt
Spectra-Physics Scanning Systems, Inc.
959 Terry Street
Eugene, OR 97402

Michael Deck
Cleanroom Software Engineering, Inc.
6894 Flagstaff Road
Boulder, CO 80302

Leslie Dent
Synopsys, Inc.
700 East Middlefield Road
Mountain View, CA 94043

David Gefen
Dept. of Comp. Info. Systems
Georgia State University
P.O. Box 4015
Atlanta, GA 30302-4015

Peter Hantos
Principal Scientist
Xerox Corporation
MS: ESAE-357
701 South Aviation Blvd.
El Segundo, CA 90245

David Kasik
Boeing Commercial Airplane Group
MS: 6H-WT
P.O. Box 3707
Seattle, WA 98124

Robert Kelsey
Digital Equipment Corp.
MS: CX02/81
301 Rockrimmon Blvd., South
Colorado Springs, CO 80919-2398

Brandon Masterson
Crane/ELDEC Corp.
30726 NE 183rd
Duvall, WA 98019

Don Moreaux
Proces Engineer
Hewlett-Packard Company
MS: 397
11311 Chinden Blvd.
Boise, ID 83714

Christopher Palermo
Fish & Richardson
2200 Sand Hill Road
Menlo Park, CA 94025

Karen Parker
Summit Design
9305 SW Gemini Drive
Beaverton, OR 97005

Steven Schurr
Attorney at Law
225 West Washington Street
Chicago, IL 60606

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

Shengru Tu
Computer Science Department
University of New Orleans
2300 Lakeshore Drive
New Orleans, LA 70148

Jeffrey Voss
Vice President
Reliable Software Technologies Corp.
21515 Ridgetop Circle
Sterling, VA 20166

Anton Webber
QA Manager
MSR Development
4619 North Street
Nacogdoches, TX 75961

Gordon Wright
American Management Systems
1455 Frazee Road
San Diego, CA 92108

Barbara Zimmer
Consulting Engineer
Hewlett-Packard Company
MS: 5MR
1501 Page Mill Road
Palo Alto, CA 94304

# Prospects for an Engineering Discipline of Software

Mary Shaw, Carnegie Mellon University

## Abstract

*Software engineering is not yet a true engineering discipline, but it has the potential to become one. Older engineering fields offer glimpses of the character software engineering might have. From these hints and an assessment of the current state of software practice, we can project some characteristics software engineering will have and suggest some steps toward an engineering discipline of software.*

*The term software engineering was coined in 1968 as a statement of aspiration–sort of a rallying cry. That year, NATO convened a workshop by that name to assess the state and prospects of software production (NATO 69). Capturing the imagination of software developers, the phrase achieved popularity during the 1970s. It now refers to a collection of management processes, software tooling, and design activities for software development. The resulting practice, however, differs significantly from the practice of older forms of engineering.*

*We begin by examining the usual practice of engineering and the way it has evolved in other disciplines. This discussion provides a historical context for assessing the current practice of software production and setting out an agenda for attaining an engineering practice.*

## About the speaker

*Mary Shaw is the Alan J. Perlis Professor of Science and Associate Dean for Professional Education at Carnegie Mellon University. She served as Chief Scientist of CMU's Software Engineering Institute. Dr. Shaw received a BA (cum laude) from Rice University and completed her Ph.D. at Carnegie Mellon.*

*Her research interests in computer science lie primarily in the areas of programming systems and software engineering, particularly software architecture, programming languages, specifications, and abstraction techniques.*

*Dr. Shaw is a Fellow of the IEEE and of the AAAS.*

**Software engineering is not yet a true engineering discipline, but it has the potential to become one. Older engineering fields suggest the character software engineering might have.**

The term "software engineering" was coined in 1968 as a statement of aspiration–a sort of rallying cry. That year, the North Atlantic Treaty Organization convened a workshop by that name to assess the state and prospects of software production. Capturing the imagination of software developers, the NATO phrase "software engineering" achieved popularity during the 1970s. It now refers to a collection of management processes, software tooling, and design activities for software development. The resulting practice, however, differs significantly from the practice of older forms of engineering.

## What is engineering?

"Software engineering" is a label applied to a set of current practices for development. But using the word "engineering" to describe this activity takes considerable liberty with the common use of that term. The more customary usage refers to the disciplined appplication of scientific knowledge to resolve conflicting constraints and requirements for problems of immediate, practical significance.

Definitions of "engineering" abound. Although details differ, they share some common clauses:

- *Creating cost-effective solutions*...Engineering is not just about solving problems; it is about solving problems with economical use of all resources, including money.
- ...*to practical problems*...Engineering deals with practical problems whose solutions matter to people outside the engineering domain–the customers.
- ...*by applying scientific knowledge*...Engineering solves problems in a particular way: by applying science, mathematics, and design analysis.
- ...*to building things*...Engineering emphasizes the solutions, which are usually tangible artifacts.
- ...*in the service of mankind.* Engineering not only serves the immediate *(continued next page)*

customer, but it also develops technology and expertise that will support the society.

Engineering relies on codifying scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice. Engineers of ordinary talent can then apply this knowledge to solve problems far faster than they otherwise could. In this way, engineering shares prior solutions rather than relying always on virtuoso problem solving.

Engineering practice enables ordinary practitioners so they can create sophisticated systems that work — unspectacularly, perhaps, but reliably. The history of development is marked by both successes and failures. The successes have often been virtuoso performances or the result of diligence and hard work. The failures have often reflected poor understanding of the problem to be solved, mismatch of solution to problem, or inadequate follow-through from design to implementation. Some failed by never working, others by overrunning cost and schedule budgets.

In current software practice, knowledge about techniques that work is not shared effectively with later projects, nor is there a large body of development knowledge organized for ready reference. Computer science has contributed some relevant theory, but practice proceeds largely independently of this organized knowledge. Given this track record, there are fundamental problems with the use of the term "software engineer."

**Routine and innovative design.** Engineering design tasks are of several kinds. One of the most significant distinctions separates routine from innovative design. Routine design involves solving familiar problems, reusing large portions of prior solutions. Innovative design, on the other hand, involves finding novel solutions to unfamiliar problems. Original designs are much more rarely needed than routine designs, so the latter is the bread and butter of engineering.

Most engineering disciplines capture, organize, and share design knowledge to make routine design simpler. Handbooks and manuals are often the carriers of this organized information. But current notations for software designs are not adequate for the task of both recording and communicating designs, so they fail to provide a suitable representation for such handbooks.

Software in most application domains is treated more often as original than routine — certainly more so than would be necessary if we captured and organized what we already know. One path to increased productivity is identifying applications that could be routine and developing appropriate support.

The current focus on reuse emphasizes capturing and organizing existing knowledge of a particular kind: knowledge expressed in the form of code. Indeed, subroutine libraries — especially of system calls and general-purpose mathematical routines — have been a staple of programming for decades. But this knowledge cannot be useful if programmers do not know about it or are not encouraged to use it. Furthermore, library components require more care in design, implementation, and documentation than similar components that are simply embedded in systems.

***Given our track record, there are fundamental problems with the use of the term "software engineer."***

Practitioners recognize the need for mechanisms to share experience with good designs. This cry from the wilderness appeared on the Software Engineering News Group, a moderated electronic mailing list:

"In Chem E, when I needed to design a heat exchanger, I used a set of references that told me what the constants were ... and the standard design equations. ...

"In general, unless I, or someone else in my [software-] engineering group, has read or remembers and makes known a solution to a past problem, I'm doomed to recreate the solution. ... I guess ... the critical difference is the ability to put together little pieces of the problem that are relatively well known, without having to generate a custom solution for every application. ...

"I want to make it clear that I am aware of algorithm and code libraries, but they are incomplete solutions to what I am describing. (There is no *Perry's Handbook for Software Engineering.*)"

This former chemical engineer is complaining that software lacks the institutionalized mechanisms of a mature engineering discipline for recording and disseminating demonstrably good designs and ways to choose among design alternatives. (*Perry's Chemical Engineering Handbook*, published by McGraw-Hill, is the standard design handbook for chemical engineering; it is about four inches thick and printed in tiny type on 8.5″ × 11″ tissue paper.)

**Model for the evolution of an engineering discipline.** Historically, engineering has emerged from ad hoc practice in two stages: First, management and production techniques enable routine production. Later, the problems of routine production stimulate the development of a supporting science; the mature science eventually merges with established practice to yield professional engineering practice. Figure 1 shows this model.

The exploitation of a technology begins with craftsmanship: A set of problems must be solved, and they get solved any which way. They are solved by talented amateurs and by virtuosos, but no distinct professional class is dedicated to problems of this kind. Intuition and brute force are the primary movers in design and construction. Progress is haphazard, particularly before the advent of good communication; thus, solutions are invented and reinvented. The transmission of knowledge between craftsmen is slow, in part because of underdeveloped communications, but also because the talented amateurs often do not recognize any special need to communicate.

Nevertheless, ad hoc practice eventually moves into the folklore. This craft stage of development sees extravagant use of available materials. Construction or manufacture is often for personal or local use or for barter, but there is little or no large-scale production in anticipation of resale. Community barn raisings are an example

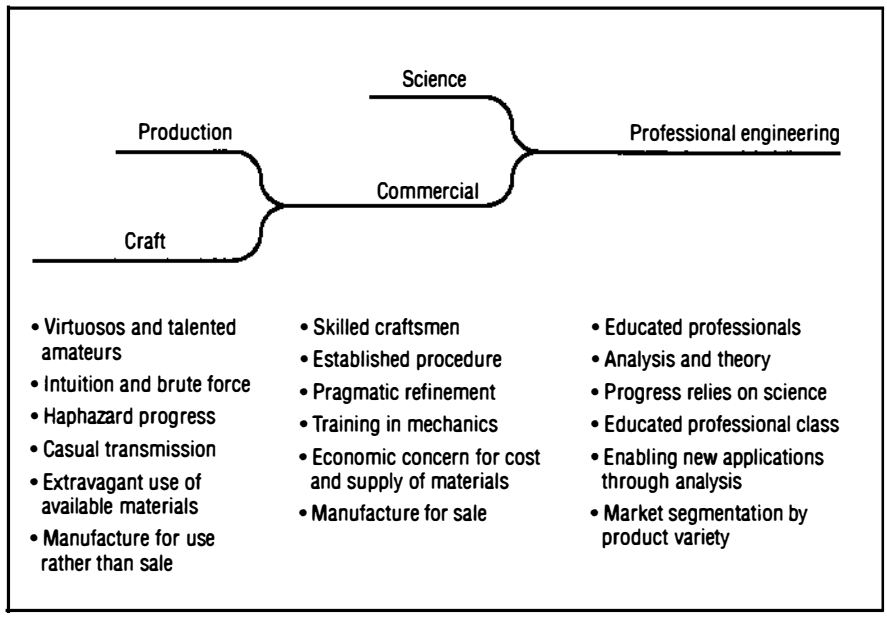



**Figure 1.** Evolution of an engineering discipline. The lower lines track the technology, and the upper lines show how the entry of production skills and scientific knowledge contribute new capability to the engineering practice.

of this stage; so is software written by application experts for their own ends.

At some point, the product of the technology becomes widely accepted and demand exceeds supply. At that point, attempts are made to define the resources necessary for systematic commercial manufacture and to marshal the expertise for exploiting these resources. Capital is needed in advance to buy raw materials, so financial skills become important, and the operating scale increases over time.

As commercial practice flourishes, skilled practitioners are required for continuity and for consistency of effort. They are trained pragmatically in established procedures. Management may not know why these procedures work, but they know the procedures *do* work and how to teach people to execute them.

The procedures are refined, but the refinement is driven pragmatically: A modification is tried to see if it works, then incorporated in standard procedure if it does. Economic considerations lead to concerns over the efficiency of procedures and the use of materials. People begin to explore ways for production facilities to exploit the technology base; economic issues often point out problems in commercial practice. Management strategies for controlling development fit at this point of the model.

The problems of current practice often stimulate the development of a corresponding science. There is frequently a strong, productive interaction between commercial practice and the emerging science. At some point, the science becomes sufficiently mature to be a significant contributor to the commercial practice. This marks the emergence of engineering practice in the sense that we know it today — sufficient scientific basis to enable a core of educated professionals so they can apply the theory to analysis of problems and synthesis of solutions.

For most disciplines, this emergence occurred in the 18th and early 19th centuries as the common interests in basic physical understandings of natural science and engineering gradually drew together. The reduction of many empirical engineering techniques to a more scientific basis was essential to further engineering progress. And this liaison stimulated further advances in natural science. "An important and mutually stimulating tie-up between natural and engineering science, a development [that] had been discouraged for centuries by the long-dominant influence of early Greek thought, was at long last consummated," wrote historian James Kip Finch.[1]

The emergence of an engineering discipline lets technological development pass limits previously imposed by relying on intuition; progress frequently becomes dependent on science as a forcing function. A scientific basis is needed to drive analysis, which enables new applications and even market segmentation via product variety. Attempts are made to gain enough control over design to target specific products on demand.

Thus, engineering emerges from the commercial exploitation that supplants craft. Modern engineering relies critically on adding scientific foundations to craft and commercialization. Exploiting technology depends not only on scientific engineering but also on management and the marshaling of resources. Engineering and science support each other: Engineering generates good problems for science, and science, after finding good problems in the needs of practice, returns workable solutions. Science is often not driven by the immediate needs of engineering; however, good scientific problems often follow from an understanding of the problems that the engineering side of the field is coping with.

The engineering practice of software has recently come under criticism for lacking a scientific basis. The usual curriculum has been attacked for neglecting mathematics[2] and engineering science.[3] Although current software practice does not match the usual expectations of an engineering discipline, the model described here suggests that vigorous pursuit of applicable science and the reduction of that science to practice *can* lead to a sound engineering discipline of software.

**Examples from traditional engineering.** Two examples make this model concrete: the evolution of engineering disciplines as demonstrated by civil and chemical engineering. The comparison of the two is also illuminating, because they have very different basic organizations.

*Civil engineering: a basis in theory.* Originally so-called to distinguish it from military engineering, civil engineering included all of civilian engineering until the middle of the 19th century. A divergence of interests led engineers specializing in other technologies to break away, and today civil engineers are the technical experts of the construction industry. They are concerned primarily with large-scale, capital-intensive construction efforts, like buildings, bridges, dams, tunnels, canals, highways, railroads, public water supplies, and sanitation. As a rule, civil-engineering efforts involve well-defined task groups that use appropriate tools and technolo-

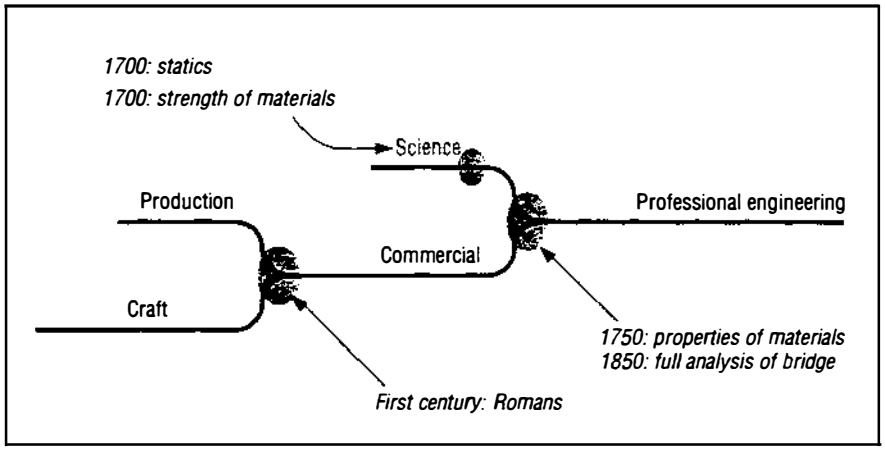



**Figure 2.** Evolution of civil engineering.

gies to execute well-laid plans.

Although large civil structures have been built since before recorded history, only in the last few centuries has their design and construction been based on theoretical understanding rather than on intuition and accumulated experience. Neither the artisans of the Middle Ages nor of the ancient world showed any signs of the deliberate quantitative application of mathematics to determine the dimensions and shapes that characterizes modern civil engineering. But even without formal understanding, they documented pragmatic rules for recurring elements. Practical builders had highly developed intuitions about statics and relied on a few empirical rules.

The scientific revolution of the Renaissance led to serious attempts by Galileo Galilei, Filippo Brunelleschi, and others to explain structures and why they worked. Over a period of about 200 years, there were attempts to explain the composition of forces and bending of a beam. However, progress was slowed for a long time by problems in formulating basic notions like force, particularly the idea that gravity could be treated as just another force like all the others. Until the basic concepts were sorted out, it was not possible to do a proper analysis of the problem of combining forces (using vector addition) that we now teach to freshmen, nor was it possible to deal with strengths of materials.

Around 1700, Pierre Varignon and Isaac Newton developed the theory of statics to explain the composition of forces and Charles Augustin de Coulomb and Louis Marie Henri Navier explained bending with the theory of strength of materials. These now provide the basis for civil engineering. By the middle of the 18th century, civil engineers were tabulating properties of materials.

The mid-18th century also saw the first attempts to apply exact science to practical building. Pope Benedict ordered an analysis of St. Peter's dome in 1742 and 1743 to determine the cause of cracks and propose repairs; the analysis was based on the principle of virtual displacement and was carried out precisely (although the model is now known to fail to account properly for elasticity). By 1850, it was possible for Robert Stephenson's Britannia Tubular Bridge over the Menai Strait between Wales and England to be subjected to a formal structural analysis.

Thus, even after the basic theories were in hand, it took another 150 years before the theory was rich enough and mature enough to have direct utility at the scale of a bridge design.

Civil engineering is thus rooted in two scientific theories, corresponding to two classical problems. One problem is the composition of forces: finding the resultant force when multiple forces are combined. The other is the problem of bending: determining the forces within a beam supported at one end and weighted at the other. Two theories, statics and strength of materials, solve these problems; both were developed around 1700. Modern civil engineering is the application of these theories to the problem of constructing buildings.

"For nearly two centuries, civil engineering has undergone an irresistible transition from a traditional craft, concerned with tangible fashioning, towards an abstract science, based on mathematical calculation. Every new result of research in structural analysis and technology of materials signified a more rational design, more economic dimensions, or entirely new structural possibilities. There were no apparent limitations to the possibilities of analytical approach; there were no apparent problems in building construction [that] could not be solved by calculation," wrote Hans Straub in his history of civil engineering.[4]

You can date the transition from craft to commercial practice to the Romans' extensive transportation system of the first century. The underlying science emerged about 1700, and it matured to successful application to practice sometime between the mid-18th century and the mid-19th century. Figure 2 places civil engineering's significant events on my model of engineering evolution.

*Chemical engineering: a basis in practice.* Chemical engineering is a very different kind of engineering than civil engineering. This discipline is rooted in empirical observations rather than in a scientific theory. It is concerned with practical problems of chemical manufacture; its scope covers the industrial-scale production of chemical goods: solvents, pharmaceuticals, synthetic fibers, rubber, paper, dyes, fertilizers, petroleum products, cooking oils, and so on. Although chemistry provides the specification and design of the basic reactions, the chemical engineer is responsible for scaling the reactions up from laboratory scale to factory scale. As a result, chemical engineering depends as heavily on mechanical engineering as on chemistry.

Until the late 18th century, chemical production was largely a cottage industry. The first chemical produced at industrial scale was alkali, which was required for the manufacture of glass, soap, and textiles. The first economical industrial process for alkali emerged in 1789, well before the atomic theory of chemistry explained the underlying chemistry. By the mid-19th century, industrial production of dozens of chemicals had turned the British Midlands into a chemical-manufacturing district. Laws were passed to control the resulting pollution, and pollution-control inspectors, called alkali inspectors, monitored plant compliance.

One of these alkali inspectors, G.E. Davis, worked in the Manchester area in the late 1880s. He realized that, although the plants he was inspecting manufactured dozens of different kinds of chemicals, there were not dozens of different

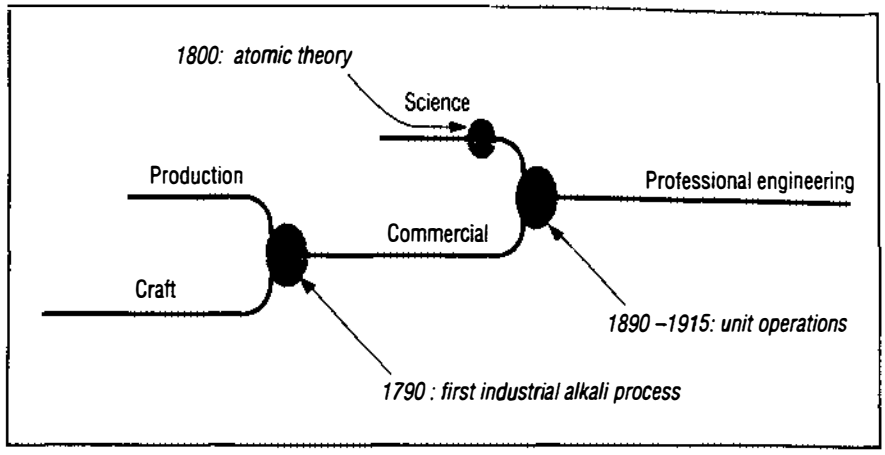



**Figure 3.** Evolution of chemical engineering.

procedures involved. He identified a collection of functional operations that took place in those processing plants and were used in the manufacture of different chemicals. He gave a series of lectures in 1887 at the Manchester Technical School. The ideas in those lectures were imported to the US by the Massachusetts Institute of Technology in the latter part of the century and form the basis of chemical engineering as it is practiced today. This structure is called *unit operations*; the term was coined in 1915 by Arthur D. Little.

The fundamental problems of chemical engineering are the quantitative control of large masses of material in reaction and the design of cost-effective industrial-scale processes for chemical reactions.

The unit-operations model asserts that industrial chemical-manufacturing processes can be resolved into a relatively few units, each of which has a definite function and each of which is used repeatedly in different kinds of processes. The unit operations are steps like filtration and clarification, heat exchange, distillation, screening, magnetic separation, and flotation. The basis of chemical engineering is thus a pragmatically determined collection of very high-level functions that adequately and appropriately describe the processes to be carried out.

"Chemical engineering as a science ... is not a composite of chemistry and mechanical and civil engineering, but a science of itself, the basis of which is those unit operations [that] in their proper sequence and coordination constitute a chemical process as conducted on the industrial scale. These operations ... are not the subject matter of chemistry as such nor of mechanical engineering. Their treatment is in the quantitative way, with proper exposition of the laws controlling them and of the materials and equipment concerned in them," the American Institute of Chemical Engineers Committee on Education wrote in 1922.[5]

This is a very different kind of structure from that of civil engineering. It is a pragmatic, empirical structure — not a theoretical one.

You can date the transition from craft to commercial practice to the introduction of the LeBlanc process for alkali in 1789. The science emerged with the British chemist John Dalton's atomic theory in the early 19th century, and it matured to successful merger with large-scale mechanical processes in the 1890s. Figure 3 places chemical engineering's significant events on my model.

## Software technology

Where does software stand as an engineering discipline? For software, the problem is appropriately an engineering problem: creating cost-effective solutions to practical problems, building things in the service of mankind.

**Information processing as an economic force.** The US computer business — including computers, peripherals, packaged software, and communications — was about $150 billion in 1989 and is projected to be more than $230 billion by 1992. The packaged-software component is projected to grow from $23.7 billion to $37.5 billion in this period, according to the Data Analysis Group's fourth-quarter 1989 forecasts. Services, including systems integration and in-house development, are not included in these figures.

Worldwide, software sales amounted to about $65 billion in 1989. This does not include the value of in-house development, which is a much larger activity. World figures are hard to estimate, but the cost of in-house software in the US alone may be in the range of $150 billion to $200 billion.[6] It is not clear how much modification after release (so-called "maintenance") is included in this figure. Thus, software is coming to dominate the cost of information processing.

The economic presence of information processing also makes itself known through the actual and opportunity costs of systems that do *not* work. Examples of costly system failures abound. Less obvious are the costs of computing that is not even tried: development backlogs so large that they discourage new requests, gigabytes of unprocessed raw data from satellites and space probes, and so on. Despite very real (and substantial) successes, the litany of mismatches of cost, schedule, and expectations is a familiar one.

**Growing role of software in critical applications.** The US National Academy of Engineering recently selected the 10 greatest engineering achievements of the last 25 years.[7] Of the 10, three are informatics achievements: communications and information-gathering satellites, the microprocessor, and fiber-optic communication. Two more are direct applications of computers: computer-aided design and manufacturing and the computerized axial tomography scan. And most of the rest are computer-intensive: the Moon landing, advanced composite materials, the jumbo jet, lasers, and the application of genetic engineering to produce new pharmaceuticals and crops.

The conduct of science is increasingly driven by computational paradigms standing on equal footing with theoretical and experimental paradigms. Both scientific and engineering disciplines require very sophisticated computing. The demands are often stated in terms of raw processing power — "an exaflop ($10^{18}$) processor with teraword memory," "a petabyte ($10^{15}$) of storage," as one article put it[8] — but the supercomputing community is increasingly recognizing development, not mere raw processing, as a critical bottleneck.

Because of software's pervasive presence, the appropriate objective for its developers should be the effective delivery of computational capability to real users in forms that match their needs. The dis-

**Table 1.**
**Significant shifts in research attention.**

| Attribute | 1960 ± 5 years: programming any-which-way | 1970 ± 5 years: programming-in-the small | 1980 ± 5 years: programming-in-the-large |
|---|---|---|---|
| Characteristic problems | Small programs | Algorithms and programming | Interfaces, management system structures |
| Data issues | Representing structure and symbolic information | Data structures and types | Long-lived databases, symbolic as well as numeric |
| Control issues | Elementary understanding of control flows | Programs execute once and terminate | Program assemblies execute continually |
| Specification issues | Mnemonics, precise use of prose | Simple input/output specifications | Systems with complex specifications |
| State space | State not well understood apart from control | Small, simple state space | Large, structured state space |
| Management focus | None | Individual effort | Team efforts, system lifetime maintenance |
| Tools, methods | Assemblers, core dumps | Programming language, compilers, linkers, loaders | Environments, integrated tools, documents |

tinction between a system's computational component and the application it serves is often very soft — the development of effective software now often requires substantial application expertise.

**Maturity of development techniques.** Our development abilities have certainly improved over the 40 or so years of programming experience. Progress has been both qualitative and quantitative. Moreover, it has taken different forms in the worlds of research and practice.

One of the most familiar characterizations of this progress has been the shift from programming-in-the-small to programming-in-the-large. It is also useful to look at a shift that took place 10 years before that, from programming-any-which-way to programming-in-the-small. Table 1 summarizes these shifts, both of which describe the focus of attention of the software research community.

Before the mid-1960s, programming was substantially ad hoc; it was a significant accomplishment to get a program to run at all. Complex software systems were created — some performed very well — but their construction was either highly empirical or a virtuoso activity. To make programs intelligible, we used mnemonics, we tried to be precise about writing comments, and we wrote prose specifications. Our emphasis was on small programs, which was all we could handle predictably.

We did come to understand that computers are symbolic information processors, not just number crunchers — a significant insight. But the abstractions of algorithms and data structures did not emerge until 1967, when Donald Knuth showed the utility of thinking about them in isolation from the particular programs that happened to implement them.

A similar shift in attitudes about specifications took place at about the same time, when Robert Floyd showed how attaching logical formulas to programs allows formal reasoning about the programs. Thus, the late 1960s saw a shift from crafting monolithic programs to an emphasis on algorithms and data structures. But the programs in question were still simple programs that execute once and then terminate.

You can view the shift that took place in the mid-1970s from programming-in-the-small to programming-in-the-large in much the same terms. Research attention turned to complex systems whose specifications were concerned not only with the functional relations of the inputs and outputs, but also with performance, reliability, and the states through which the system passed. This led to a shift in emphasis to interfaces and managing the programming process.

In addition, the data of complex systems often outlives the programs and may be more valuable, so we learned that we now have to worry about integrity and consistency of databases. Many of our programs (for example, the telephone switching system or a computer operating system) should *not* terminate; these systems require a different sort of reasoning than do programs that take input, compute, produce output, and terminate. In systems that run indefinitely, the sequence of system states is often much more important than the (possibly undesirable) termination condition.

The tools and techniques that accompanied the shift from programming-any-which-way to programming-in-the-small provided first steps toward systematic, routine development of small programs; they also seeded the development of a science that has matured only in the last decade. The tools and techniques that accompanied the shift from programming-in-the-small to programming-in-the-large were largely geared to supporting groups of programmers working together in orderly ways and to giving management a view into production processes. This directly supports the commercial practice of development.

Practical development proceeded to large complex systems much faster than the research community did. For example, the Sage missile-defense system of the 1950s and the Sabre airline-reservation system of the 1960s were successful interactive systems on a scale that far exceeded the maturity of the science. They appear to have been developed by excellent engineers who understood the requirements well and applied design and development methods from other (like electrical) engineering disciplines. Modern development methodologies are management procedures intended to guide large numbers of developers through similar disciplines.

The term "software engineering" was introduced in 1968 to name a conference

convened by NATO to discuss problems of software production.[9] Despite the label, most of the discussion dealt with the challenge of progressing from the craft stage to the commercial stage of practice. In 1976, Barry Boehm proposed the definition of the term as "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them."[10] This definition is consistent with traditional definitions of engineering, although Boehm noted the shortage of scientific knowledge to apply.

Unfortunately, the term is now most often used to refer to life-cycle models, routine methodologies, cost-estimation techniques, documentation frameworks, configuration-management tools, quality-assurance techniques, and other techniques for standardizing production activities. These technologies are characteristic of the commercial stage of evolution — "software management" would be a much more appropriate term.

**Scientific basis for engineering practice.** Engineering practice emerges from commercial practice by exploiting the results of a companion science. The scientific results must be mature and rich enough to model practical problems. They must also be organized in a form that is useful to practitioners. Computer science has a few models and theories that are ready to support practice, but the packaging of these results for operational use is lacking.

*Maturity of supporting science.* Despite the criticism sometimes made by software producers that computer science is irrelevant to practical software, good models and theories *have* been developed in areas that have had enough time for the theories to mature.

In the early 1960s, algorithms and data structures were simply created as part of each program. Some folklore grew up about good ways to do certain sorts of things, and it was transmitted informally. By the mid-1960s, good programmers shared the intuition that if you get the data structures right, the rest of the program is much simpler. In the late 1960s, algorithms and data structures began to be abstracted from individual programs, and their essential properties were described and analyzed.

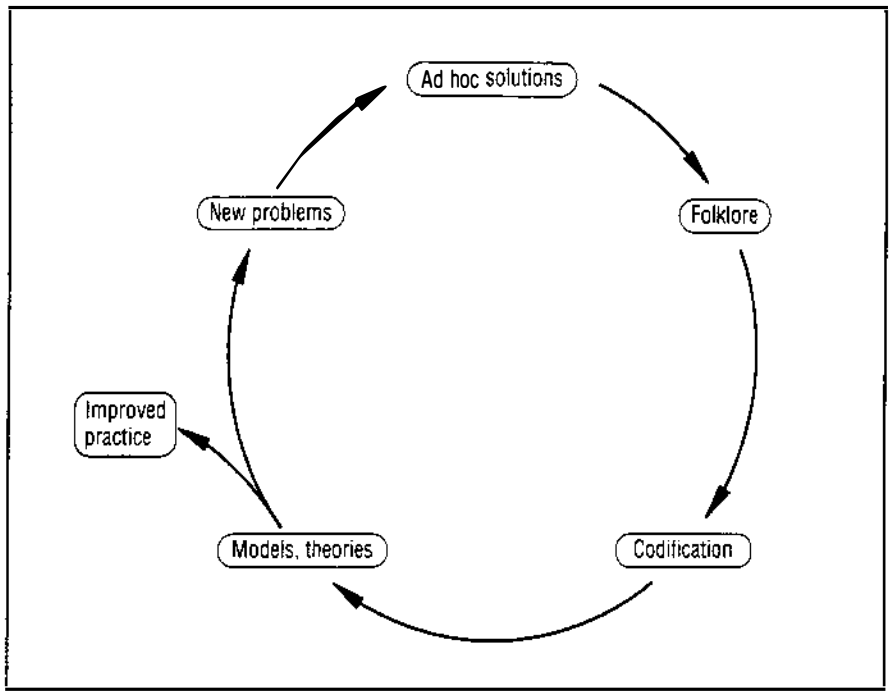



**Figure 4.** Cycle of how good software models develop as a result of the interaction between science and engineering.

The 1970s saw substantial progress in supporting theories, including performance analysis and correctness. Concurrently, the programming implications of these abstractions were explored: abstract-data-type research dealt with such issues as:

- Specifications: abstract models and algebraic axioms.
- Software structure: bundling representation with algorithms.
- Language issues: modules, scope, and user-defined types.
- Information hiding: protecting the integrity of information not in the specification.
- Integrity constraints: invariants of data structures.
- Composition rules: declarations.

Both sound theory and language support were available by the early 1980s, and routine good practice now depends on this support.

Compiler construction is another good example. In 1960, simply writing a compiler at all was a major achievement; it is not clear that we really understood what a higher level language was. Formal syntax was first used systematically for Algol-60, and tools for processing it automatically (then called compiler compilers, but now called parser generators) were first developed in the mid-1960s and made practical in the 1970s. Also in the 1970s, we started developing theories of semantics and types, and the 1980s have brought significant progress toward the automation of compiler construction.

Both of these examples have roots in the problems of the 1960s and became genuinely practical in the 1980s. It takes a good 20 years from the time that work starts on a theory until it provides serious assistance to routine practice. Development periods of comparable length have also preceded the widespread use of systematic methods and technologies like structured programming, Smalltalk, and Unix, as Sam Redwine and colleagues have shown.[11] But the whole field of computing is only about 40 years old, and many theories are emerging in the research pipeline.

*Interaction between science and engineering.* The development of good models within the software domain follows this pattern:

We engineers begin by solving problems any way we can. After some time, we distinguish in those ad hoc solutions things that usually work and things that do not usually work. The ones that do work enter the folklore: People tell each other about them informally. As the folklore becomes more and more systematic, we codify it as written heuristics and rules of procedure. Eventually, that codification becomes crisp enough to support models and theories, together with the associated mathematics. These can then help improve practice,

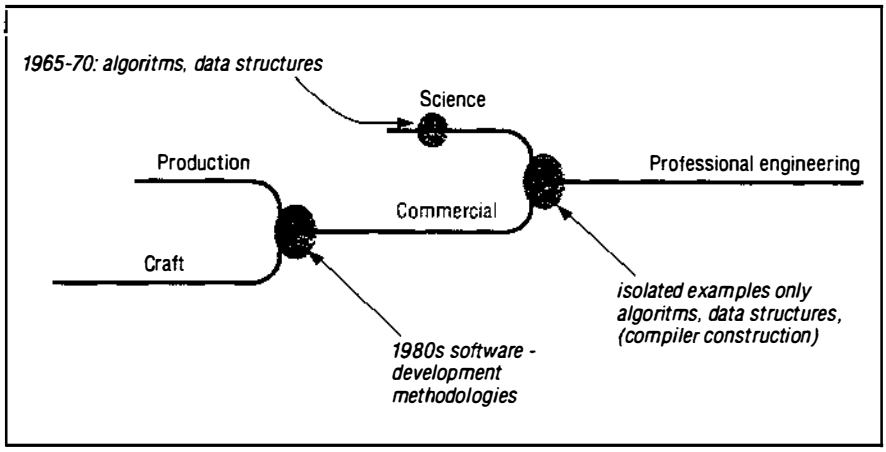



**Figure 5.** Evolution of software engineering.

and experience from that practice can sharpen the theories. Furthermore, the improvement in practice let us think about harder problems — which we first solve ad hoc, then find heuristics for, eventually develop new models and theories for, and so on. Figure 4 illustrates this cycle.

The models and theories do not have to be fully fleshed out for this process to assist practice: The initial codification of folklore may be useful in and of itself.

This progression is illustrated in the use of machine language for control flow in the 1960s. In the late 1950s and the early 1960s, we did not have crisp notions about what an iteration or a conditional was, so we laid down special-purpose code, building each structure individually out of test and branch instructions.

Eventually, a small set of patterns emerged as generally useful, generally easy to get right, and generally at least as good as the alternatives. Designers of higher level languages explicitly identified the most useful ones and codified them by producing special-purpose syntax. A formal result about the completeness of the structured constructs provided additional reassurance.

Now, almost nobody believes that new kinds of loops should be invented as a routine practice. A few kinds of iterations and a few kinds of conditionals are captured in the languages. They are taught as control concepts that go with the language; people use them routinely, without concern for the underlying machine code.

Further experience led to verifiable formal specifications of these statements' semantics and of the programs that used them. Experience with the formalization in turn refined the statements supported in programming languages. In this way, ad hoc practice entered a period of folklore and eventually matured to have conventional syntax and semantic theories that explain it.

**Where is software?** Where, then, does current software practice lie on the path to engineering? It is still in some cases craft and in some cases commercial practice. A science is beginning to contribute results, and, for isolated examples, you can argue that professional engineering is taking place. (Figure 5 shows where software practice fits on my model.)

That is not, however, the common case.

There are good grounds to expect that there will eventually be an engineering discipline of software. Its nature will be technical, and it will be based in computer science. Although we have not yet matured to that state, it is an achievable goal.

The next tasks for the software profession are

- to pick an appropriate mix of short-term, pragmatic, possible purely empirical contributions that help stabilize commercial practice and
- to invest in long-term efforts to develop and make available basic scientific contributions.

The profession must take five basic steps on its path to becoming a true engineering discipline:

**Understand the nature of expertise.** Proficiency in any field requires not only higher order reasoning skills but also a large store of facts together with a certain amount of context about their implications and appropriate use. Studies have demonstrated this across a wide range of problem domains, including medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others, as Herbert Simon described in the paper "Human Experts and Knowledge-Based Systems" presented at the 1987 IFIP Working Group 10.1 Workshop on Concepts and Characteristics of Knowledge-Based Systems.

An expert in a field must know about 50,000 chunks of information, where a chunk is any cluster of knowledge sufficiently familiar that it can be remembered rather than derived. Furthermore, in domains where there are full-time professionals, it takes no less than 10 years for a world-class expert to achieve that level of proficiency.[11]

Thus, fluency in a domain requires content and context as well as skills. In the case of natural-language fluency, E.D. Hirsch has argued that abstract skills have driven out content; students are expected (unrealistically) to learn general skills from a few typical examples rather than by a "piling up of information"; and intellectual and social skills are supposed to develop naturally without regard to the specific content.[12]

However, Hirsch wrote, specific information is important at all stages. Not only are the specific facts important in their own right, but they serve as carriers of shared culture and shared values. A software engineer's expertise includes facts about computer science in general, software design elements, programming idioms, representations, and specific knowledge about the program of current interest. In addition, it requires skill with tools: the language, environment, and support software with which this program is implemented.

Hirsch provided a list of some 5,000 words and concepts that represent the information actually possessed by literate Americans. The list goes beyond simple vocabulary to enumerate objects, concepts, titles, and phrases that implicitly invoke cultural context beyond their dictionary definitions. Whether or not you agree in detail with its composition, the list and accompanying argument demonstrate the need for connotations as well as denotations of the vocabulary.

Similarly, a programmer needs to know not only a programming language but also the system calls supported by the environment, the general-purpose libraries, the application-specific libraries, and how to combine invocations of these definitions effectively. The programmer must

**Table 2.**
**Cost distributions for the thee ways to get a piece of information.**

| Method | Infrastructure cost | Initial-learning cost | Cost of use in practice |
|---|---|---|---|
| Memory | Low | High | Low |
| Reference | High | Low | Medium |
| Derivation | Medium-high | Medium | High |

be familiar with the global definitions of the program of current interest and the rules about their use. In addition, a developer of application software must understand application-area issues.

Simply put, the engineering of software would be better supported if we knew better what specific content a software engineer should know. We could organize the teaching of this material so useful subsets are learned first, followed by progressively more sophisticated subsets. We could also develop standard reference materials as carriers of the content.

**Recognize different ways to get information.** Given that a large body of knowledge is important to a working professional, we as a discipline must ask how software engineers should acquire the knowledge, either as students or as working professionals. Generally speaking, there are three ways to get a piece of information you need: You can remember it, you can look it up, or you can derive it. These have different distributions of costs, as Table 2 shows.

Memorization requires a relatively large initial investment in learning the material, which is then available for instant use.

Reference materials require a large investment by the profession for developing both the organization and the content; each student must then learn how to use the reference materials and then do so as a working professional.

Deriving information may involve ad hoc creation from scratch, it may involve instantiation of a formal model, or it may involve inferring meaning from other available information. To the extent that formal models are available, their formulation requires a substantial initial investment. Students first learn the models, then apply them in practice. Because each new application requires the model to be applied anew, the cost in use may be very high.[13]

Each professional's allocation of effort among these alternatives is driven by what he has already learned, by habits developed during that education, and by the reference materials available. Today, general-purpose reference material for software is scarce, although documentation for specific computer systems, languages, and applications may be extensive. Even when documentation is available, however, it may be underused because it is poorly indexed or because developers have learned to prefer fresh derivation to use of existing solutions. The same is true of subroutine libraries.

Simply put, software engineering requires investment in the infrastructure cost — in creating the materials required to organize information, especially reference material for practitioners.

**Encourage routine practice.** Good engineering practice for routine design depends on the engineer's command of factual knowledge and design skills and on the quality of reference materials available. It also depends on the incentives and values associated with innovation.

Unfortunately, computer-science education has prepared developers with a background that emphasizes fresh creation almost exclusively. Students learn to work alone and to develop programs from scratch. They are rarely asked to understand software systems they have not written. However, just as natural-language fluency requires instant recognition of a core vocabulary, programming fluency should require an extensive vocabulary of definitions that the programmer can use familiarly, without repeated recourse to documentation.

Fred Brooks has argued that one of the great hopes for software engineering is the cultivation of great designers.[14] Indeed, innovative designs require great designers. But great designers are rare, and most designs need not be innovative. Systematic presentation of design fragments and techniques that are known to work can enable designers of ordinary talent to produce effective results for a wide range of more routine problems by using prior results (buying or growing, in Brooks's terms) instead of always building from scratch.

It is unreasonable to expect a designer or developer to take advantage of scientific theories or experience if the necessary information is not readily available. Scientific results need to be recast in operational form; the important information from experience must be extracted from examples. The content should include design elements, components, interfaces, interchange representations, and algorithms. A conceptual structure must be developed so the information can be found when it is needed. These facts must be augmented with analysis techniques or guidelines to support selection of alternatives that best match the problem at hand.

A few examples of well-organized reference materials already exist. For example, the summary flowchart of William Martin's sorting survey[15] captured in one page the information a designer needed to choose among the then-current sorting techniques. William Cody and William Waite's manual for implementing elementary mathematical functions[16] gives for each function the basic strategy and special considerations needed to adapt that strategy to various hardware architectures.

Although engineering has traditionally relied on handbooks published in book form, a software engineers' handbook must be on line and interactive. No other alternative allows for rapid distribution of updates at the rate this field changes, and no other alternative has the potential for smooth integration with on-line design tools. The on-line incarnation will require solutions to a variety of electronic-publishing problems, including distribution, validation, organization and search, and collection and distribution of royalties.

Simply put, software engineering would benefit from a shift of emphasis in which both reference materials and case studies of exemplary software designs are incorporated in the curriculum. The discipline must find ways to reward preparation of material for reference use and the development of good case studies.

**Expect professional specializations.** As software practice matures toward engineering, the body of substantive technical knowledge required of a designer or developer continues to grow. In some areas, it has long since grown large enough to

require specialization — for example, database administration was long ago separated from the corresponding programming. But systems programming has been resistant to explicit recognition of professional specialties.

In the coming decade, we can expect to see specialization of two kinds:

- internal specialization as the technical content in the core of software grows deeper and
- external specialization with an increased range of applications that require both substantive application knowledge and substantive computing knowledge.

Internal specialties are already starting to be recognizable for communications, reliability, real-time programming, scientific computing, and graphics, among others. Because these specialties rely critically on mastery of a substantial body of computer science, they may be most appropriately organized as postbaccalaureate education.

External specialization is becoming common, but the required dual expertise is usually acquired informally (and often incompletely). Computational specializations in various disciplines can be supported via joint programs involving both computer science and the application department; this is being done at some universities.

Simply put, software engineering will require explicit recognition of specialties. Educational opportunities should be provided to support them. However, this should not be done at the cost of a solid foundation in computer science and, in the case of external specialization, in the application discipline.

**Improve the coupling between science and commercial practice.** Good science is often based on problems underlying the problems of production. This should be as true for computer science as for any other discipline. Good science depends on strong interactions between researchers and practitioners. However, cultural differences, lack of access to large, complex systems, and the sheer difficulty of understanding those systems have interfered with the communication that supports these interactions.

Similarly, the adoption of results from the research community has been impeded by poor understanding of how to turn a research result into a useful element of a production environment. Some companies and universities are already developing cooperative programs to bridge this gap, but the logistics are often daunting.

Simply put, an engineering basis for software will evolve faster if constructive interaction between research and production communities can be nurtured. ❖

## Acknowledgments

This article benefited from comments by Allen Newell, Norm Gibbs, Frank Friedman, Tom Lane, and the other authors of articles in this special issue. Most important, Eldon Shaw fostered my appreciation for engineering. Without his support, this work would not have been possible, so I dedicate this article to his memory.

This work was supported by the US Defense Dept. and a grant from Mobay Corp.

## References

1. J.K. Finch, *Engineering and Western Civilization*, McGraw-Hill, New York, 1951.
2. E.W. Dijkstra, "On the Cruelty of Really Teaching Computing Science," *Comm. ACM*, Dec. 1989, pp. 1,398-1,404.
3. D.L. Parnas, "Education for Computing Professionals," *Computer*, Jan. 1990, pp. 17-22.
4. H. Straub. *A History of Civil Engineering: An Outline from Ancient to Modern Times*, MIT Press, Cambridge, Mass., 1964.
5. F.J. van Antwerpen, "The Origins of Chemical Engineering," in *History of Chemical Engineering*, W.F. Furter, ed., American Chemical Society, Washington, D.C., 1980, pp. 1-14.
6. Computer Science and Technology Board, National Research Council, *Keeping the US Computer Industry Competitive*, National Academy Press, Washington, D.C., 1990.
7. National Academy of Engineering, *Engineering and the Advancement of Human Welfare: 10 Outstanding Achievements 1964-1989*, National Academy Press, Washington, D.C., 1989.
8. E. Levin, "Grand Challenges to Computational Science," *Comm. ACM*, Dec. 1989, pp. 1,456-1,457.
9. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 1968*, P. Naur and B. Randell, eds., Scientific Affairs Div., NATO, Brussels, 1969.
10. B.W. Boehm, "Software Engineering," *IEEE Trans. Computers*, Dec. 1976, pp. 1,226-1,241.
11. S.T. Redwine et al., "DoD-Related Software Technology Requirements, Practices, and Prospects for the Future," Tech. Report P-1788, Inst. Defense Analyses, Alexandria, Va., 1984.
12. E.D. Hirsch, Jr., *Cultural Literacy: What Every American Needs to Know*, Houghton Mifflin, Boston, 1989.
13. M. Shaw, D. Giuse, and R. Reddy, "What a Software Engineer Needs to Know I: Vocabulary," tech. report CMU/SEI-89-TR-30, Carnegie Mellon Univ., Pittsburgh, Aug. 1989.
14. F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Information Processing 86*, pp. 1,069-1,076.
15. W.A. Martin, "Sorting," *ACM Computing Surveys*, Dec. 1971, pp. 147-174.
16. W.J. Cody, Jr., and W.M. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

**Mary Shaw** is a professor of computer science at Carnegie Mellon University, where she has been on the faculty since 1971. From 1984 to 1987, she was chief scientist at the Software Engineering Institute, with which she still has a joint appointment. Her primary research interests are programming systems and software engineering, particularly abstraction techniques and language tools for developing and evaluating software.

Shaw received a BA in mathematics from Rice University and a PhD in computer science from Carnegie Mellon University. She is a member of the IEEE Computer Society, ACM, New York Academy of Sciences, and Sigma Xi. She also serves on the National Research Council's Computer Science and Technology Board, IEEE Technical Committee on Software Engineering, and IFIP Working Group 2.4 (System Implementation Languages).

Address questions about this article to the author at Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA 15213-3890; Internet shaw@cs.cmu.edu.

# A Cleanroom for Your Mother

Dr. Charles B. Engle, Jr., Florida Institute of Technology

## Abstract

*Remember when your mother told you that you could not go out and play until you had a clean room? Well, this talk is not about that! Instead, its an exploration of the Cleanroom Software Engineering (CSE) methodology as originally proposed by Dr. Harlan Mills and as it has evolved today.*

*Cleanroom Engineering develops software of certified reliability under statistical quaity control in a pipeline of increments that accumulate into the specified software product. In the Cleanroom process, no program debugging is permitted before independent statistical usage testing of the increments as they accumulate into the final product. The Cleanroom development process provides rigorous methods of software specification, development, and certification, through which disciplined software engineering teams are capable of producing low- or zero-defect software of arbitrary size and complexity. Such engineering discipline is not only capable of producing highly reliable software, but also certifying the reliability of the software as specified.*

*Cleanroom is really based on a few simple principles that are not unique to cleanroom. The principles bring the real power to Cleanroom.*

*The Cleanroom principles can be implemented in many ways. There is not "orthodox" Cleanroom, since each of the principles is adopted and adapted to the current situation, and further, implemented in the most appropriate manner. This allows Cleanroom to be implemented under any existing development process at any maturity level, although, clearly, the more fully* that the principles are embraced and *the more mature the process, the better for the consumer of the future.*

*Cleanroom projects have succeeded to varying degrees, but all have shown increased productivity with decreased defect rates. Four new Cleanroom projects will be detailed. Some of the ongoing projects will be described.*

*You know, perhaps this talk was about what your mother meant. Perhaps you shouldn't go out and play without a Cleanroom.*

## About the speaker

*Dr. Charles B. Engle, Jr., is a professor, lecturer, and consultant in Software Engineering, Ada, and Cleanroom. He has been involved in Clean room since 1989, working directly with the inventor of the techniques, Dr. Harlan Mills. He has practiced these principles as a consultant in several large organizations and helped introduce them into the curriculum at his university.*

*Dr. Engle is the Chair of the Computer Science program at the Florida Institute of Technology. Prior to joining the faculty at Florida Tech, Dr. Engle was the Deputy Program Manager and Site Director of the Software Engineering Institute JPO, and simultaneously a Senior Member of the Technical Staff in the education program. Dr. Engle was also an Associate Professor of Computer Science at the United States Military Academy at West Point.*

*He is currently on leave of absence in an IPA position in Washington, D.C., serving as the Technical Director to the Center for Software at the Defense Information Systems Agency (DISA) and concurrently as Chief, Ada Joint Program Office.*

# A Cleanroom for Your Mother

Charles B. Engle, Jr.

Pacific Northwest Software Quality Conference
September, 1995





## Presentation Overview

- Introduction
- Cleanroom Overview
- Cleanroom Definition and Principles
- Cleanroom Specification
- Cleanroom Design and Implementation
- Cleanroom Certification
- Cleanroom Management

# Introduction

## What is Cleanroom Software Engineering?

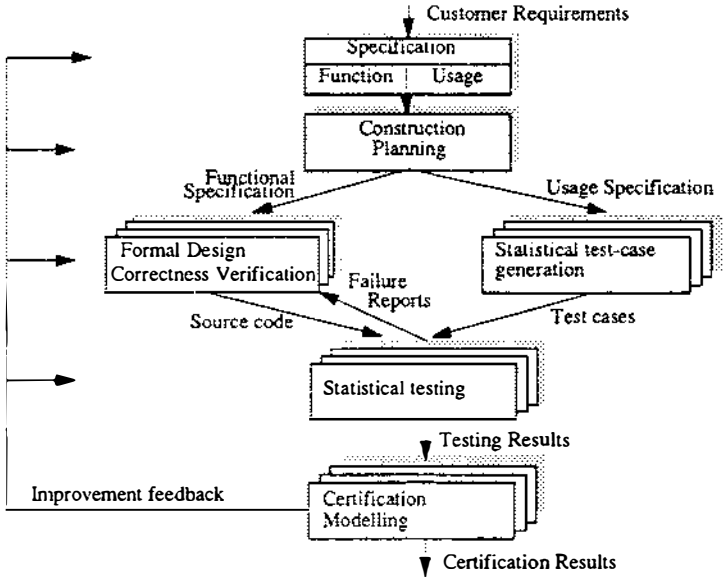
"The Cleanroom engineering process develops software of certified reliability under statistical quality control in a pipeline of increments with functional verification but no program debugging permitted before independent statistical usage testing of the increments."

-Harlan D. Mills





# Cleanroom Overview



# Cleanroom Definitions and Principles

## Base Principle - Intellectual Control

Cleanroom principles and practices should provide intellectual control by operating in these dimensions

- Process Driven Development
- Incremental Development
- Precise Specification of Behavior
- Formal Design and Verification
- Statistical Certification Testing

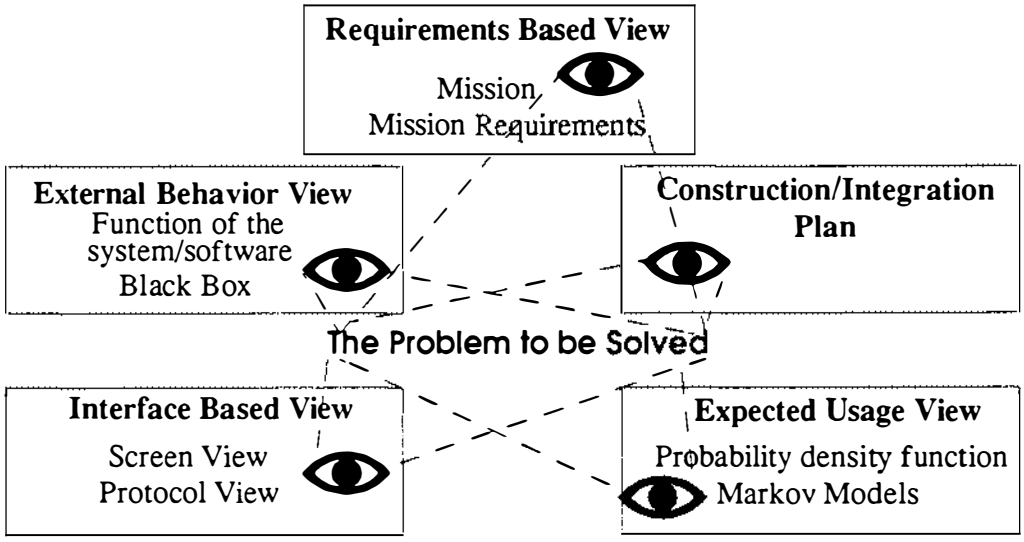






# Cleanroom Specifications

## Specifications Should Entail Multiple Views



## Historical Perspective

• The Vasa debacle



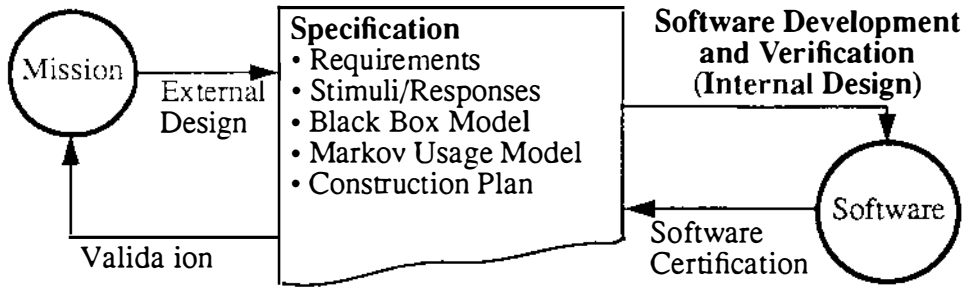


## Cleanroom Design and Implementation

**Design and Implementation => Development Team**

**Mission:**

Given a set of specifications which are to be implemented in software, design and verify the behaviors, data, and processes that correctly implement the specifications.



## Development Team Principles

The development team uses engineering principles, methods and processes which:

- Emphasize the simple and avoid the complex.
- Strive for intellectual control over the design and the project.
- Recognize the full potential of people.
- Provide people with the intellectual tools they require.





## Box Structure Hierarchy

| | | |
|---|---|---|
| External View | Black Box | Implementation-free description of behavior (responses) in terms of historical sequences of stimuli |
| Data View | State Box | Intermediate description of behavior (responses) in terms of state data usage, where state data is used as a surrogate for stimulus histories |
| Process View | Clear Box | Internal description of behavior (responses) in terms of state data usage and control structure usage of program parts |

- Box structures allow for three independent and unique views of a system

## Fundamental Principles of Box Structures

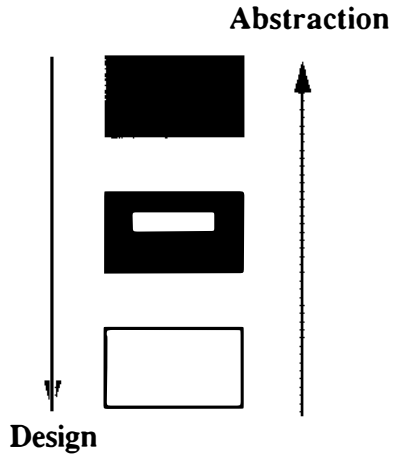
Every system exhibits black box behavior.

Every black box can be designed as a state box.

Every state box can be designed as a clear box possibly introducing new black boxes.

Every clear box can be abstracted into a state box.

Every state box can be abstracted into a black box.










## Effective Use of Box Structures

- Referential Transparency
- Transaction Closure
- State Migration
- Identification of Application Objects
- Identification of Common Services Objects
- Efficient Verification
- A Correct Design Trail
- Separation of Concerns

# Cleanroom CertificationSoftware Testing

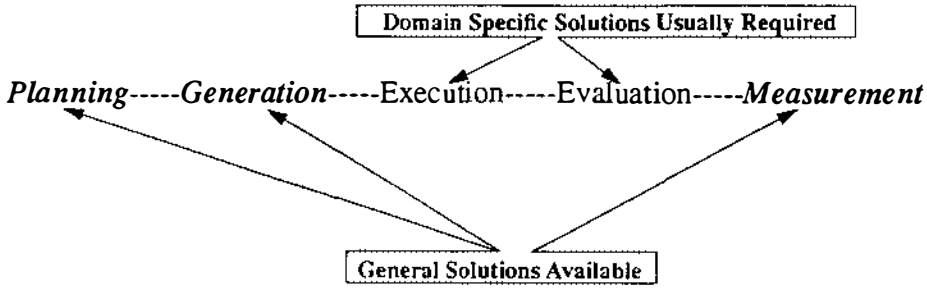
**Definitions:**

*Software testing* is the process of executing a software system to determine the extent to which it is correct with respect to its specification

*Failures* occur when the observed behavior deviates from the intended behavior

*Faults* are the bug(s) in the code that cause software failures

**Phases:**








# Cleanroom Certification

| Certification Testing | Traditional Testing |
|---|---|
| • Tests are engineered | • Tests are crafted |
| • Testers create models of test cases | • Testers create individual test cases |
| • Test outcomes can be inexpensively simulated before testing | • Simulation is expensive or impossible |
| • Testing decisions are based on objective, scientific measures | • Testing decisions are based on subjective intuition |
| • Testing results in quantifiable product quality certification | • No product certification possible |

## Cleanroom Management

- Team Approach
  - All project work is performed as negotiated team assignments
  - Teams have different responsibilities
- Frequent reviews
  - Spread knowledge, build the team spirit, find defects
- Process driven development
  - The process is the basis for planning, directing and assignment of tasks
- Incremental development
  - Teams are allocated to increments
  - No big-bang integration at the end







## Managing Cleanroom Projects

Adherence to the process is key

- There will be many temptations not to follow process, especially when there are deadlines, or it is the first project
- Following the process is what achieves the results
- Never cut down on reviews

Effort profile will be different

- More time in specification
- Less time in design
- Less time in test
- Less time correcting failures

Teams should be empowered to organize and direct themselves

## Summary

Cleanroom is based upon key principles

- Process Driven Development
- Incremental Development
- Precise Specification of Behavior
- Formal Design and Verification
- Statistical Certification Testing

Cleanroom is focused upon three major technical areas

- Specification
- Design and Implementation
- Certification

# SOFTWARE PROCESS ASSESSMENTS AROUND THE WORLD

Gordon Wright
American Management Systems, Inc.
1455 Frazee Road, Suite 315
San Diego, CA 92108
(619) 683-5640
gordon_wright@mail.amsinc.com

**Key Words** - Assessment, Software Engineering Institute, Software Process Improvement, Requirements Management, Capability Maturity Model, Software Quality Assurance, Configuration Management, Software Formal Inspections.

**Abstract** - This paper describes the process of performing Software Engineering Institute (SEI) Software Process Assessments (SPAs) for a large international company with software development centers located around the world. It describes the SPA process, the vagaries of performing SPAs in foreign countries, typical assessment findings and recommendations of foreign-based commercisoftware development organizations, and software process improvement steps after the SPA.

**Gordon Wright** - Mr. Wright has over 20 years in the software development field, primarily in MIS development and software project management. He has worked in private industry as well as for the U.S. Navy. As Director of Software Process Improvement for American Management Systems, Inc., he leads Software Process Assessments for commercial and Department of Defense clients and develops and teaches software process improvement training courses. He was a charter member of the Software Engineering Process Office at the Naval Command, Control & Ocean Surveillance Center R&D Division (NRaD) in 1989 where he implemented a software cost estimation process. He also participated in implementing other software development processes including metrics and formal inspections. He participated in the development and presentation of NRaD's Software Project Management course and currently teaches Software Process Improvement, Software Project Management, Software Estimation, Software Formal Inspections and Software Tools.

# SOFTWARE PROCESS ASSESSMENTS AROUND THE WORLD

Gordon Wright
American Management Systems, Inc.
1455 Frazee Road, Suite 315
San Diego, CA 92108
(619) 683-5640
gordon_wright@mail.amsinc.com

**Abstract -** This paper describes the process of performing Software Engineering Institute (SEI) Software Process Assessments (SPAs) for a large international company with software development centers located around the world. It describes the SPA process, the vagaries of performing SPAs in foreign countries, typical assessment findings and recommendations of foreign-based commercial software development organizations, and software process improvement steps after the SPA.

**I. Introduction** - Major corporations around the world are adopting formal process improvement practices in order to stay current with competition. Organizations around the world have adopted the SEI software CMM as their model for software process improvement. One particular international bank has set out to have every one of its software development sites implement software process improvement activities. The corporate goal is for all software development sites of 40 people or more to improve their software development process maturity level and be able to prove it. Each major software development activity is to be assessed by the SEI assessment method. Approximately two years later they will be reassessed and must show improvement. This paper addresses the assessments AMS has conducted for that organization as well as for other commercial and Department of Defense related companies and agencies. AMS has conducted SPAs in South America, Saudi Arabia, India, Philippines, Hong Kong, Taiwan, Japan, Australia, Canada, England and Singapore.

International Banking - The banking communities around the world face a variety of problems that are not so different from the U.S. banking community: quick turnaround promotions, expanding services, rebundling current services, and keeping up with banking technology. Additionally, banks in foreign countries are faced with the challenge of coming on-line to the world economy. For long established banking communities, such as those of Europe, matriculating into the world economy is not as difficult as for the not so advanced countries such as some Latin American countries or the third world countries. All these pressures add to the pressure already put on software development groups to implement error-free products as expeditiously as possible.

**II. Software Process Assessments** - In 1991, AMS was one of nine original vendors selected for the commercialization of the Carnegie-Mellon University Software Engineering Institute's (SEI) Software Process Assessment (SPA) methodology. In the past four years, AMS has performed over 50 assessments of software development groups ranging from 20-person to 400-person in size within corporations with 100 to 50,000 employees.

The Software Process Improvement program at SEI exists to improve the process of developing and maintaining software. The goal is to accelerate the maturity of software engineering as a practice and thus improve the quality and productivity of software development and maintenance. Toward that end, SEI has developed a method of assessing an organization's software development process maturity and provides a set of recommended actions which, if taken, will assist in improving the process within the organization.

Our SEI-trained assessors work with an organization's professional staff to provide a rigorous, well-defined assessment process that encompasses one week of training and one week of assessment (see Figures 1 and 2). The assessment team is comprised of four to seven members of the organization and two SEI-trained assessors. The week of training focuses on assessment principles, how to conduct an SEI assessment, determining an organization's software maturity level, change management and the concepts of the SEI software Capability Maturity Model (CMM) (see References 1 and 2). The training includes role playing and rehearsals for conducting interviews and facilitating group discussions. Additionally, during this first week, Project Leaders selected to participate in the assessment answer the Capability Maturity Model questionnaire. The questionnaire responses are the basis for the Project Leader Script that is utilized for the first round of Project Leader interviews on Monday of the assessment week.

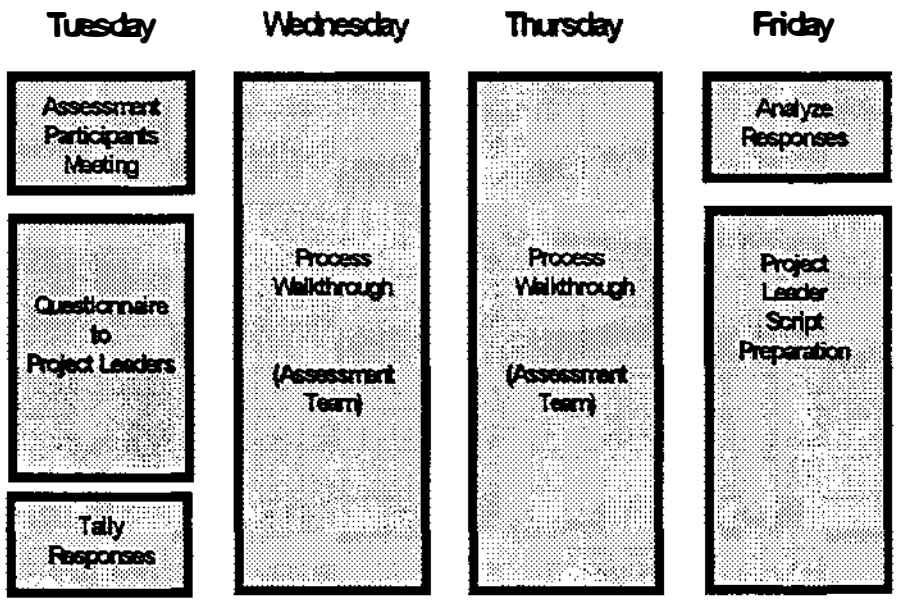



Figure 1. SPA Training Week

During the assessment week, the Project Leaders are interviewed on Monday and then presented with the preliminary findings on Wednesday. On Tuesday the Functional Area Representatives (FARs) meet and discuss issues and problems they have had or

are currently experiencing relative to the functional area they represent. Typical functional areas include:

- Requirements Definition and Analysis
- Preliminary & Detailed Software Design
- Coding and Unit Test
- User Acceptance Test

During these sessions, the FARs and Project Leaders also provide information on the strengths of the organization. On Thursday, the draft findings are presented to the Project Leaders and the FARs. This is their opportunity to comment and provide feedback on the Assessment Team's findings. After these dry runs, the final findings briefing is finalized and presented to Senior Management on Friday.

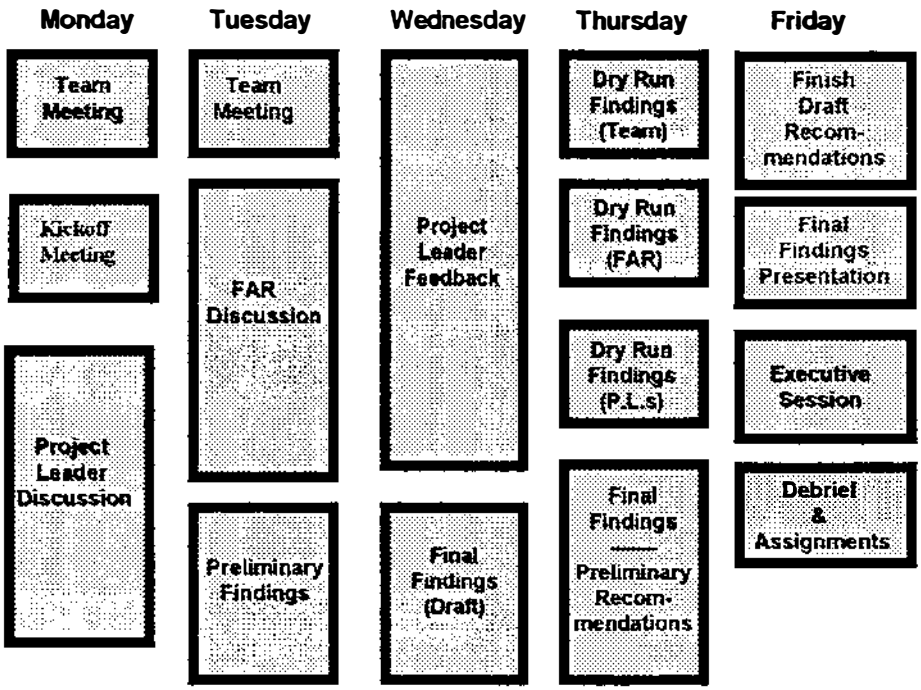



Figure 2. SPA Assessment Week

Software Process Assessments in Foreign Countries - There are some considerations that assessors must address for conducting assessments in foreign countries that are not present when doing U.S. assessments. As true in any business endeavor in a foreign country, there are cultural considerations such as schedules, timeliness, holidays and various local customs. In some countries, the software developers and assessment team members do not all speak English.

If the assessment is to be conducted in a foreign language, translation of materials must be addressed before the assessment process begins. The SEI CMM Questionnaire must be translated if all the participating software project leaders do not

read English. During the on-site training and assessment, a translator, usually supplied by the host organization, is often required. For instance, English-Spanish translators were necessary for most of the South American countries and a Chinese translator was necessary in Taiwan. The translator may have to translate for the U.S. assessors as briefings are presented, for any questions asked by the U.S. assessors of the organization's software practitioners, and for translation of written material produced during the assessment. If any translation is required, then generally as a minimum, the assessment kick-off briefing, final findings briefing, and scripted questions used during the interviews all have to be translated and a translator must be present for all interviews. The translating however, does not greatly impact the assessment schedule. The only activities that require additional time because of the translating are the briefings. The interviews are conducted by the host members of the assessment team.

One impact of the language barrier is that sometimes the "getting acquainted" process between trained assessor and the group's assessors take longer than usual. It is important for the team to achieve a strong rapport as soon as possible because of the rigor of the assessment process.

Configuration management of interview and briefing materials becomes especially important during this time as there are two versions being developed concurrently during the two week period. For instance, in Spanish countries, the final briefing will be done in Spanish and an English version will also be generated. Keeping the two versions in sync has been known to extend the already late evening schedules. The only translation done in advance of the two week visit is the CMM Questionnaire.

Differences in languages are not the only difficulties that have arisen while conducting SPAs in foreign countries. A transportation labor strike in India almost had our assessors heading straight for the airport. One morning on their way to work, their cab was surrounded by menacing strikers. The cab driver managed to convince the angry crowd that his passengers were foreigners who were not unsympathetic to their cause but in fact were not aware of the local labor controversy. They arrived at work safely. However, later that morning that cab suffered broken windows as well as some other damage.

Eating can also be hazardous to an assessors health. Stomach ailments due to exotic food as well as the local water are not uncommon. It is tempting to try as much of the local food as you can but experience has proven that in some countries it is best to stay to hotel food or restaurants that cater to foreigners. However, this is not always possible because local members of the organization will sometime graciously act as tour guides during weekends and evenings and take you to their favorite places

The differences in time zones also presents difficulties. There is the obvious difficulty in adjusting to times that are 12 to 18 hours different. There is also the difficulty of making phone calls during the preparation and follow-up phases. Difficulties in phone communications to Spanish speaking organizations in South America is compounded by the lack of English speaking receptionists and the long lunches they often enjoy. We do a lot of communicating by FAX.

Difficulties of Foreign Based Software Developers - While our assessors encounter some problems, developers in foreign countries have more serious difficulties. One of

the typical difficulties that the foreign organizations have is obtaining knowledge about what is going on in the world of software development, specifically in the U.S. with organizations such as the SEI. The absence of technical books and software trade periodicals hinder them in learning the new methodologies being implemented. It is not only difficult to obtain software engineering books and tools in some countries but it is even difficult to know what is available. This may not always be so difficult for members of international organizations but for local organizations it can be a real problem. However, even if there were easy access to these materials, many developers do not speak English or read English. To overcome this, many organizations offer English classes to employees. The classes are free but usually have to be taken on their own time. However, even with language difficulties, some organizations are very advanced in maturity and strive to establish and maintain an environment of continuous process improvement.

**III. Comparison of Results of Foreign Country SPAs to U.S. SPAs** - SEI licensed vendors are required to submit the results of their assessments to the SEI. Each year at the Software Engineering Process Group Conference the SEI presents summary data on assessments. Table 1 shows a comparison of the results of our assessments with recent assessment data compiled by the SEI. The results of assessments we have conducted for commercial companies in both the U.S. and in foreign countries shows that the maturity level of commercial companies is lower than that of most US DOD companies. Generally, assessment results show that the results for DOD companies and agencies is better than for the commercial organizations. This is because the DOD companies have been told to improve their software development processes using the SEI CMM as a yardstick or not be awarded any new contracts. DOD companies are currently being driven to be compliant with at least level 2 (Repeatable) Key Process Areas and possibly be at level 3 (Defined) by 1997.

Table 1 - Summary of AMS SPA Findings

| CMM Level | SEI Data | AMS Data | | |
|---|---|---|---|---|
| | | DOD Vendor | Commercial/ International | Total |
| 1 - Initial | 75% | 33% | 69% | 60% |
| 2 - Repeatable | 16% | 50% | 23% | 29% |
| 3 - Defined | 8% | 17% | 8% | 11% |
| 4 - Managed | 0% | 0% | 0% | 0% |
| 5 - Optimizing | 0.5% | 0% | 0% | 0% |

The commercial world is more competitive than ever for a variety of reasons. One of the foremost reasons is the advent of the global economy. Industries such as banking now have to compete on an international scale as well as a local or national scale. This increased competition is driving banks to reduce development time for new products and services at an ever increasing pace. The increased emphasis on improved

productivity as a means to reduce development time is perceived as critical in maintaining and expanding their customer base.

Generally speaking, commercial companies did not adopt the SEI CMM until after the DOD community. Many are now actively adopting the SEI CMM or some other quality framework (e.g., ISO 9000) in order to establish and attain software process improvement goals. Their incentives may not be directly imposed by an outside agency as in the case of DOD contractors but rather as a means to improve their business posture.

Some development organizations who have embarked on successful software process improvement efforts have documented results on their increased productivity through increased quality, improved planning and tracking mechanisms, and better defined software development processes (see Reference 3). Some of our client organizations claim improved user relations have resulted from improved processes through better requirements definition and management techniques as well as improved formal testing techniques.

**IV. Summary of Assessment Findings** - Our findings show that the foreign developers experience many of the same problems as U.S. software development organizations - poor requirements management, poorly defined software management practices, little or no software quality assurance or configuration management practices in the early project stages. Not only are some organizations not performing SQA or SCM but it is not unusual for software developers to not have a basic understanding of the roles and responsibilities of SQA and SCM. A summary of results of the most common CMM KPA related findings resulting from AMS-assisted SPAs are shown in Table 2. The summary of findings shown do not reflect all software development weaknesses observed during our assessments but only those deemed to be the major issues.

Requirements Management - The most common requirements management problem is the lack of a documented agreement on which to base the project. Often, managers are required to estimate the schedule and budget for a project before anyone has had an opportunity to evaluate the requirements. Furthermore, even if requirements are agreed to before project inception, changes to requirements during development are not tracked and are not always communicated to all affected parties. This leads to products that do not function as the user intended as well as other major defects.

Project Planning - Inconsistent planning processes are very common. Not only are they inconsistent, they are not documented and are not based on any formal principle. Many software project managers do not use any explicit method for estimating size, cost and schedule of a project. Within an organization there may be some basic assumption of how to measure the size of a product, but often that assumption is erroneous. Some development groups equate size of a software product with effort. If you ask them how big they think a software program will be, they reply in terms of resources such as personmonths, or personmonths and schedule vs. in terms of product size. Many of these managers have an

awareness of lines of code or function points but have not had training or any first-hand experience with these size metrics.

Table 2 - Summary of AMS SPA Findings

| | Number of Findings Relative to CMM KPAs | | | |
|---|---|---|---|---|
| **Repeatable Level Findings** | **Govt. / DOD/ DOD Vendor** | **Commercial/ International** | **Total** | **% of Org with Finding** |
| Requirements Management | 4 | 17 | 21 | 49% |
| Project Planning | 6 | 23 | 29 | 67% |
| Project Tracking & Oversight | 3 | 15 | 18 | 42% |
| Subcontractor Management | 1 | | 1 | 2% |
| Software Quality Assurance | 6 | 18 | 24 | 56% |
| Software Configuration Management | 3 | 13 | 16 | 37% |
| | | | | |
| **Defined Level Findings** | | | | |
| Organization Process Focus | 2 | 2 | 4 | 9% |
| Organization Process Definition | 5 | 4 | 9 | 21% |
| Training Program | 9 | 15 | 24 | 56% |
| Integrated Software Management | 1 | 1 | 2 | 5% |
| Software Product Engineering | 5 | 10 | 15 | 35% |
| Intergroup Coordination | 2 | 8 | 10 | 23% |
| Peer Reviews | 3 | 4 | 7 | 16% |
| | | | | |

Project Tracking and Oversight - Status meetings are the most common form of formal project tracking and oversight. The intentions of these meetings are good but often do not result in anyone knowing quantifiably the real status of a project in terms of progress, productivity, or realistic completion dates. There often is no mechanism to ensure that all affected parties will be informed of meetings or that they will be conveyed the results of such meetings. Many of these meetings do not have an agenda but are based on some assumptions of what will be discussed.

Subcontractor Management - Most of the organizations assessed do not do subcontracting in the sense of assigning the development of a deliverable to an outside vendor. Some organizations hire contract personnel to work side by side with internal developers. In these cases, the contractors are incorporated as an integral part of the development process. Thus, as shown in Table 2, there is limited data for this KPA.

Software Quality Assurance - There are some SQA findings that are endemic to many international software organizations. For instance, many software development organizations believe they practice good SQA because they perform a physical audit (a.k.a. Physical Configuration Audit) at the end of a

project. For these organizations, the use of SQA methods across the entire development cycle is not a familiar concept. The lack of effective SQA processes in such organizations is evident from the lack of compliance with any existing standards or policies, the amount of rework that is often required, the frequent mismatch of final product vs. requirements and lateness of deliverables.

Software Configuration Management - Many international software developers often have code checkout procedures but have not heard of nor understand the concept of configuration management of other software products, e.g., Requirements Specifications, Test Plans, or Test Data. The issue of test data management is a common finding. Developers often cannot replicate errors found in previous tests because their test databases have changed or been modified by a user or another developer.

Training - While management always agree on the importance of training, a common finding is that software developers receive little software technical or software management training. Training is specified throughout the CMM KPAs. In fact there are 49 training related attributes in the CMM. As can be seen from Table 2, most organizations are deficient in training. Software developers are often expected to learn their roles and responsibilities through some form of osmosis. Senior managers will often cite company related training that personnel receive but very seldom is any of that training related to a developer's day-to-day job or to any aspect of software engineering. There are many examples of training that software personnel do not receive. For instance, very seldom do developers have a clear idea of the roles and responsibilities of SQA or Configuration Management. This leads to misunderstandings and often needless friction between developers and supporting functions. Another prime example is that in many international software development organizations, Project Leaders seldom receive any software formal project management training.

Peer Reviews - Some organizations are beginning to hold peer reviews and software formal inspections but do not collect data from the reviews or inspections. One South American software development group that is essentially operating at the Defined level, with some minor exceptions, has been conducting peer reviews for over three years. However, they have never recorded the data that would help them isolate the cause of recurring problems. At their reviews, they focus only on uncovering and fixing defects and have no mechanism to feed lessons learned back into the development process.

Non-CMM KPA Related Findings - In addition to the findings that map directly to the CMM KPAs, a number of issues that do not map as directly to the KPAs are experienced by many organizations. The three most prevalent of these other findings are:

- Standards, Guidelines, Practices, Procedures
- Communication
- Testing

There are others such as risk management and maintenance but the three issues noted above are the most common sources of problems relative to non-CMM KPA findings. It must be noted here that these areas are addressed in the CMM KPAs but assessment teams often want these findings emphasized.

Standards, Guidelines, Practices, Procedures - Some organizations need standards and guidelines such as planning, CM and SQA. Others have standards, guidelines, procedures, etc., but there are often no mechanisms in place, such as SQA, to ensure that they are followed. Furthermore, it is not unusual for developers to not know that they exist or where to find them if they do exist.

Communication - It is a very common finding that policies and procedures are sometimes mandated but not effectively communicated to all affected people. Another problem in many organizations is that decisions are made and previous decisions modified relative to delivery schedules and requirements that are not communicated to all affected people. This leads to misunderstandings regarding schedules and subsequently to poor morale.

Testing - Many testing related findings have to do with lack of comprehensive testing environments and lack of configuration management of test data. Developers are often faced with working long hours because test data has been lost or compromised. Extra hours are also often the result of poor planning. Several projects may be entering the test phase at the same time putting constraints on available resources. This often results in inadequate system testing which in turn results in the User Acceptance Testing function actually performing system testing.

**V. Sample Assessment Findings, and Recommendations** - Part of the last day of the assessment is devoted to examining the findings and developing recommendations. The number of findings are usually narrowed to five to seven issues. As mentioned above, these findings are not necessarily restricted to the CMM Key Process Areas. The recommendations for the specific findings are fairly broad at the time of the final findings briefing. Shown below are some samples of findings, consequences and recommendations related to Requirements Management, Project Planning, and SQA. These samples are from several assessments and are not necessarily related.

Requirements Management

Findings:

- Requirements frequently are not clearly defined, verifiable and testable
- Requirements for each work order are not quantified nor is each one tracked individually
- There is no formal documented process to control changes to requirements throughout the project

Consequences:

- User expectations are not met
- Delays to product delivery
- Excessive overtime required, employee morale suffers
- Rework impacts other projects and resources

Recommendations:

- Analyze the causes of misunderstood requirements
- Implement a consistent method for establishing and communicating initial requirements and changes to requirements
- Establish a process to ensure that changes are reflected in all deliverable items in a timely manner

Project Planning

Findings:

- Procedures do not exist for estimating size, cost and schedule and planning of project activities
- Estimates are based on individual experience

Consequences:

- Project estimates are not reliable
- Difficult to plan personnel assignments
- Difficult to plan resources such as hardware, test environments, disk space, etc.
- Difficult to plan coordination of projects
- Lack of consistent method of estimating among projects

Recommendations:

- Investigate and identify current estimation and tracking methods used both inside and outside the organization
- Document and implement standard estimation, planning and tracking methods
- Improve project risk analysis, contingency planning, and resource allocation planning

Software Quality Assurance

Findings:

- SQA activities are not performed for all phases of the projects
- SQA activities are not consistent across projects

Consequences:

- Lack of adherence to standards and procedures may not be detected
- Quality of work products may be impacted
- Possible inconsistencies of the products between different phases

- Projects become dependent upon key people
- Lack of management visibility into non-adherence to standards and procedures

Recommendations:

- Establish SQA function
- Define minimum standards for SQA to be applied on all software development efforts

There are General Recommendations as well as specific recommendations provided in the final briefing. Some sample General Recommendations include:

- Continue mapping the Group's Technology Process to the CMM
- Establish a software engineering training plan
- Establish a software engineering technical library
- Assign individuals to research specific areas of software engineering technology to:
  - disseminate information and sources
  - conduct technical workshops

**VI. Strengths found During Assessments** - In addition to the findings that address areas that need improvement, assessments also identify the strengths of an organization. All organizations have strengths related to software development, personnel and management. Several organizations have begun working on defining and documenting their software development processes in the form of a Software Development Life Cycle (SDLC). The SDLC addresses the overall development process as well as organizational interfaces, quality assurance and standard practices.

Some organizations have independently identified the need for improved quality and have instituted improved requirements management processes and formal software quality assurance practices. One organization assessed at the Initial level had developed a very comprehensive set of software quality assurance guidelines and was about to implement it on some pilot projects.

Some foreign organizations we have assessed have in fact been continuously implementing software process improvement initiatives for several years. An organization in the Asia-Pacific region has been keeping and applying process metrics for all phases of their development process for years. One South American organization has been working on software process improvements since 1986. They have a very clearly defined development process that they follow, review and refine. That organization missed being completely compliant at the Defined level because their peer review process did not utilize metrics and feedback techniques to improve their processes. They were completely compliant with all other Repeatable and Defined KPAs. That organization is especially noteworthy because they did not know about the SEI and the CMM until 1992.

A common strength in many of the foreign organizations is that the software developers have a strong sense of commitment to their organizations and to each other.

Consequently, it is common practice for these developers to work long hours and to share a strong sense of pride in their work. The developers who worked the longest hours were usually the most eager to learn about software process improvement and how to improve their processes.

**VII.** **After the Software Process Assessment** - We work with the assessed group to document the findings and recommendations in the Final Findings Report. This report is submitted to the SEI. The software development group must then develop an Action Plan that provides details on how the recommendations will be implemented. The Action Plan is implemented typically over 18 to 24 months. After implementation of the Action Plan, another assessment is conducted to determine the state of their software processes. Figure 3 depicts the software process improvement cycle espoused through the SEI assessments. This cycle shows the steps from the first stages of awareness through the assessment and eventually to reassessment.

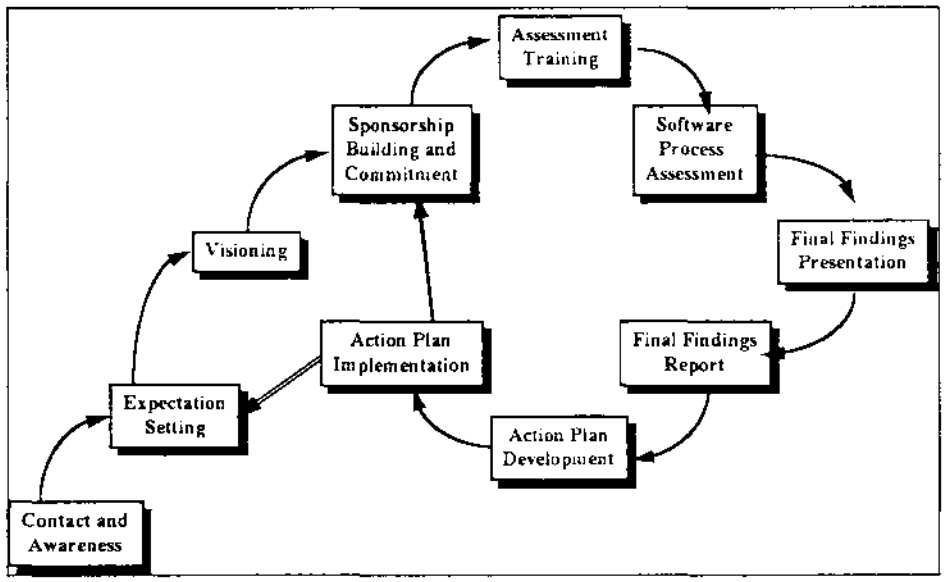



Figure 3. Software Process Improvement Cycle

**VIII.** **Conclusion** - While there are difficulties in conducting assessments in foreign countries, our experience indicates that software developers in foreign countries are eager to learn and adopt new initiatives and management technology. At the end of these assessments, the group's trained assessors are usually very eager to get started implementing some of the new methodologies they learned.

Our experiences with foreign clients proves that the use of SEI's software Capability Maturity Model by organizations throughout the world continues to grow. While there are difficulties in supporting these international organizations, we find their enthusiasm extremely rewarding and continue to support all of the development groups as much as we are able given the difficulties of working together over long distances.

References

1. Capability Maturity Model for Software, Version 1.1, SEI-93-TR-24, February 1993.
2. Key Practices of the Capability Maturity Model, Version 1.1, SEI-93-TR-25, February 1993.
3. Benefits of CMM-Based Software Process Improvement: Initial Results, CMU/SEI-94-TR-13, Herbsleb, James, et al, August 1994.

# Finding, Qualifying, and Managing Software Subcontractors within a Large Engineering Project

## A Synopsis for the Pacific Northwest Software Quality Conference

**by Jim Nielsen, Motorola, Satellite Communications Division**

Motorola's Satellite Communications Division is responsible for the Iridium® project, an ambitious large scale telecommunications engineering project. This is a project with an aggressive schedule, a sizable software content, a need for unprecedented quality, and strict reliability requirements. Most of the software content of the project requires significant domain expertise which can only be found in outside suppliers, so a large percentage of the software component is being subcontracted to outside developers.

In order to minimize risk to the project schedule, to ensure adherence to a common process model, and to achieve congruence between our requirements and the product features, all potential suppliers must undergo a formal evaluation of their capability. This evaluation is one of several decision gates in the supplier selection process. If a supplier does not have a satisfactory internal process and does not have the capability to improve that process within the project time period, they are not considered qualified as a potential software supplier. Although there are companies who have similar evaluation programs for software suppliers, the large number of potential suppliers involved with this project gave us an opportunity to examine the state of software process on a multifaceted level.

Our software supplier process evaluations were conducted in a manner very similar to an SEI evaluation, but on a much smaller scale. The traits that our evaluations share with an SEI assessment are: a focus of capability maturity, a goal of process improvement, the SEI model as a basis, an interview/discussion format, and the use of trained assessors. In addition, we incorporated several aspects of the Quality Systems Review, such as interviewing senior management and gauging the quality of the development platforms and disaster recovery plans.

The supplier evaluations spend two days on-site using a three member team as opposed to five days with an SEI assessment using a five or six member team. The supplier evaluations interview two or three project leaders and twelve to twenty functional area representatives whereas an SEI assessment will interview four or more project leaders and up to thirty-six FAR's.

When the evaluation is completed, the evaluation team will meet to compare notes and craft their findings. The findings are presented to the supplier's management team with a final rating of Not Qualified, Qualified, or Fully Qualified. This is often followed by an executive briefing which will discuss the findings and answer questions. Frequently, we establish process improvement requirements of a potential supplier in order to receive a contract. For those subcontractors who win contracts with Motorola, we conduct regular

follow-up evaluations. These may be done to ensure compliance with recommendations made during an evaluation or to just monitor progress on a long-term project.

Although the identities of our suppliers and specific information on our findings must remain confidential, the results from the evaluations of over thirty three potential suppliers from around the world have yielded some interesting observations. We have found very little difference in capability between small, medium, and large organizations, and we found little difference in capability of companies based on their type of enterprise. While a very small organization may lack a written process, they usually have an unwritten informal process which is widely understood and encultured. The companies with the greatest difficulty appeared to be those who have been undergoing reductions in force which have reduced process focus along with the staff.

Most of the suppliers which we evaluated were extremely cooperative in the effort, realizing that they had a lot to gain from the process. Since this was done at Motorola expense to ensure the capability of suppliers, most of the findings and recommendations were well-received.

Our suppliers were not the only one to gain from this effort. We learned much about how companies of different sizes and composition were able to achieve excellence in their domain areas. The information collected from the supplier evaluation effort will provide a baseline for such activities in future. All sectors of Motorola are now conducting these types of supplier evaluations when necessary. Our findings contribute to the improvement of the subcontractor management process within the company and by spreading the word, we can also assist others in the same effort. In this way, we can all achieve greater cycle time reduction, higher quality, and more effective products in the years ahead.

# Biography

## Jim Nielsen

Jim Nielsen is currently manager of the Tools and Technology Group with the Satellite Communications Division of Motorola, Inc., in Chandler, AZ. The Satellite Communications Division is responsible for the Iridium® project. While at SATCOM, he has been responsible for conducting most of the evaluations of potential software subcontractors, subcontractor process management, and is a member of the Software Engineering Process Group. He has also been a member of several teams which have conducted internal SEI Software Process Self-Assessments. His current duties include data management and the evaluation and selection of engineering tools for the division.

Mr. Nielsen has been with Motorola since 1981, spending most of those years working for the Computer Group at their design center in Beaverton, OR. Prior to that, he spent eight years in software development with various organizations.

Mr. Nielsen has a Bachelor's and Master's degree from the University of Chicago, and a Bachelor's degree from Portland State University in Portland, OR.

Jim Nielsen can be reached at Motorola, Inc., Satellite Communications Division, 2501 S Price Rd, Chandler, AZ 85248-2899; 602-732-6034 (office), 602-732-3837 (fax); email p26649@email.mot.com

MOTOROLA
Satellite Communications Division

# Finding, Qualifying, and Managing Software Subcontractors

Jim Nielsen
Motorola, Inc.
Satellite Communications Division





MOTOROLA
Satellite Communications Division

## The Iridium® Project

- Aggressive Schedule
- Stringent Quality Standards
- Reliability Requirements
- Significant Software Content







MOTOROLA
Satellite Communications Division

## Strategy

- Determine what we can do ourselves
- Use COTS where possible
- Reuse software where possible
- Locate recognized experts to build the rest





MOTOROLA
Satellite Communications Division

## Criteria for Determining Best Suppliers

- Domain Expertise
- Business Analysis
- Software Process Assessment

MOTOROLA
Satellite Communications Division

## Evaluation Matrix

| Criteria | | Weight | Vendor A Rating | Vendor A Points | Vendor B Rating | Vendor B Points | Vendor C Rating | Vendor C Points | Comments |
|---|---|---|---|---|---|---|---|---|---|
| Supplier Evaluation Findings (FQ/Q/NQ) | | N/A | | | | | | | |
| Business Analysis Findings (Pass/Fail) | | N/A | | | | | | | |
| Management Issues | | | | | | | | | |
| Issue #1 | | 25 | | | | | | | |
| Issue #2 | | 20 | | | | | | | |
| Issue #3 | | 20 | | | | | | | |
| Issue #4 | | 15 | | | | | | | |
| Issue #5 | | 20 | | | | | | | |
| | TOTAL | 100 | | | | | | | |
| Technical Issues | | | | | | | | | |
| Issue #1 | | 25 | | | | | | | |
| Issue #2 | | 10 | | | | | | | |
| Issue #3 | | 20 | | | | | | | |
| Issue #4 | | 20 | | | | | | | |
| Issue #5 | | 25 | | | | | | | |
| | TOTAL | 100 | | | | | | | |
| Software Development Plan | | | | | | | | | |
| Item #1 | | 10 | | | | | | | |
| Item #2 | | | | | | | | | |
| a. Sub-item #1 | | 5 | | | | | | | |
| b. Sub-item #2 | | 5 | | | | | | | |
| Item #3 | | 5 | | | | | | | |
| Item #4 | | 10 | | | | | | | |
| Item #5 | | 10 | | | | | | | |
| | TOTAL | 100 | | | | | | | |




MOTOROLA
Satellite Communications Division

## Process Evaluation Examines

- Process maturity
- Schedule risks
- Estimation risks
- Disaster recovery risks
- Staffing size and capability
- Management commitment and support




MOTOROLA
Satellite Communications Division

## Supplier Evaluation and SEI Self-Assessment Commonalities

- Uses SEI model as basis
- Assesses process maturity
- Uses interview/discussion format
- Uses trained assessors




MOTOROLA
Satellite Communications Division

## Supplier Evaluation and SEI Self-Assessment Differences

- 2 days on-site
- 3 member team
- 2-3 project leaders
- 12-20 functional area reps
- no feedback sessions
- senior management interviews
- qualified/not qualified rating
- no pre-evaluation questionnaire

- 5 day on-site
- 5+ member team
- 5+ project leaders
- 16-24 functional area reps
- PL/FAR feedback sessions
- no management interviews
- SEI level rating
- pre-assessment questionnaire

MOTOROLA
Satellite Communications Division

## Team Composition

- Team leader from process group
- Second member from contracting software group
- Third member from related area or process group
- All must be trained in evaluation procedures





MOTOROLA
Satellite Communications Division

## Assessor Training

**Prerequisites**

- General software background
- Basics of SEI model

**Objectives**

- Understand how SEI model applies to us
- Be able to serve on assessor team



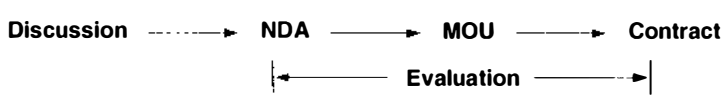




MOTOROLA
Satellite Communications Division

## Timing of Supplier Evaluation

- High likelihood of potential contract
- After Non-Disclosure Agreement in place
- Before issuance of contract








MOTOROLA
Satellite Communications Division

## Evaluation Results by Evaluation Type

| | Fully Qualified | Qualified | Not Qualified |
|---|---|---|---|
| Full Evaluation | 4 | 17 | 2 |
| Internal Motorola | 1 | 5 | - |
| COTS | 3 | 1 | - |

Total Suppliers Evaluated 33

## Evaluation Results by Supplier Origin

| | Fully Qualified | Qualified | Not Qualified |
|---|---|---|---|
| Europe | 1 | 4 | - |
| Asia | 2 | 1 | 1 |
| North America | 5 | 18 | 1 |






## Evaluation Results by Supplier Size

| | Fully Qualified | Qualified | Not Qualified |
|---|---|---|---|
| Less than 20 employees | - | 4 | - |
| 20 - 400 employees | 4 | 15 | 2 |
| 401 - 1000 employees | 2 | 1 | - |
| More than 1000 employees | 2 | 3 | - |






## Evaluation Results by Supplier Type

| | Fully Qualified | Qualified | Not Qualified |
|---|---|---|---|
| Satellite | 3 | 3 | - |
| Telephony | 2 | 8 | - |
| Network | 1 | 7 | 1 |
| Simulation & Test | 1 | 3 | - |
| General | 1 | 2 | 1 |






**SATCOM SW Supplier Maturity Scale (for Internal use only)**

| SMM Rating | Description |
|---|---|
| 0<br>Not Qualified | Software Process Assessment not performed, or subcontractor found to be not qualified. |
| 1<br>Qualified Low | SW Methodology somewhat defined, however not consistently followed or implemented, schedule risk exists, no quantitative measurements in place, some project tracking mechanism in place, some uncertainty in domain experience. Missing a maximum of two of the following key elements: Rqmts Mgmt, SW Project Planning, SW Project Tracking & Oversight, SW Subcontracts Mgmt, SQA, SCM. Considered "qualified" on SATCOM supplier evaluation. |
| 2<br>Qualified | SW Methodology defined and followed by the majority of the organization, SW process supported by mgmt, obvious commitment by people interviewed, some quantitative measurements in place, moderate schedule risk, risk mitigation is considered and addressed in project plans. Recognized for their domain experience. The following key elements must be part of the subcontractor's software process: Rqmts Mgmt, SW Project Planning, SW Project Tracking & Oversight, SW Subcontracts Mgmt, SQA, SCM. (Equivalent to SCI level 2) |

MOTOROLA
Satellite Communications Division

| SATCOM SW Supplier Maturity Scale (for Internal use only) | |
|---|---|
| **SMM Rating** | **Description** |
| 3<br>Qualified High | Fully documented life-cycle Software methodology in place and supported by entire organization. The following key elements are in place: Rqmts Mgmt, SW Project Planning, SW Project Tracking & Oversight, SW Subcontracts Mgmt, SQA, SCM. Quantitative measurements in place. Continuous improvements plans, based on quantitative measurements not in place. Considered "quatified" on SATCOM supplier evaluation. (Considered a high SEI level 2) |
| 4<br>Fully Qualified | Institutionalized Software Process. The following key elements must either be in the process of being implemented, or part of the subcontractor's existing software process: Organization Process Focus, Organization Process Definition, Training Program, Integrated Software Management, Software Process Product Engineering, Intergroup Coordination, and Peer reviews. Continuous improvements plans in place which are based on quantitative measurements. SW processes are quantitatively understood and controlled using defined measures. Considered "Fully Qualified" in SATCOM supplier evaluation. (Considered an SEt level 3 or higher organization.) |




MOTOROLA
Satellite Communications Division

## Comparison of Evaluation Results By Scale

| Using Qualified/Not Qualified Scale | | |
|---|---|---|
| **Not Qualified** | **Qualified** | **Fully Qualified** |
| 2 | 23 | 8 |

| Using SMM Scale | | | | |
|---|---|---|---|---|
| **Rating 0** | **Rating 1** | **Rating 2** | **Rating 3** | **Rating 4** |
| 1 | 6 | 11 | 9 | 6 |






MOTOROLA
Satellite Communications Division

## Most Frequent Strengths

- Software process
- Quality staff
- Organizational focus on quality product
- Customer focus and contact
- Configuration management
- Training
- Tools and technology




MOTOROLA
Satellite Communications Division

## Most Frequent Opportunities

- Process improvement
- Insufficient data collection and use of metrics
- Lack of adequate training
- Inconsistent internal communication
- Unclear vision and management support
- Unprepared for growth

MOTOROLA
Satellite Communications Division

## Supplier Evaluation Agenda

**Day 1**

08:00 - 08:30 Pre-assessment meeting with coordinator
08:30 - 09:30 Meeting with management to introduce team
09:30 - 10:00 Meeting with General Manager (private with team)
10:00 - 10:15 Break
10:15 - 10:45 Meeting with Engineering Manager (private with team)
10:45 - 11:15 Meeting with Chief SW Engineer (private with team)
11:15 - 12:00 Meeting with Project Leader #1 (private with team)
12:00 - 01:00 Lunch
01:00 - 01:45 Meeting with Project Leader #2 (private with team)
01:45 - 02:45 Group Interview -- QA, CM, and Release (3-5 eng.)
02:45 - 03:00 Break
03:00 - 04:00 Group Interview -- Test and Integration (3-5 eng.)
04:00 - 05:00 Group Interview -- Code and Unit Test (3-5 eng.)




MOTOROLA
Satellite Communications Division

## Supplier Evaluation Agenda

Evening of Day 1 - Meet to discuss and craft initial findings

**Day 2**

08:00 - 08:30 Meeting with coordinator to make adjustments
08:30 - 09:30 Group Interview -- Requirements Spec (3-5 eng.)
09:30 - 10:00 Meeting with QE/QA Manager
10:00 - 12:00 Team recaps notes and findings, reviews documents
12:00 - 01:00 Lunch
01:00 - 03:00 Crafting of final findings
03:00 - 04:00 Presentation of final findings to senior management
04:00 - 04:30 Meeting with General Manager (optional)






MOTOROLA
Satellite Communications Division

## Evaluation Follow-up

- Written report based on findings
- List of specific improvement actions, if necessary
- Improvement plan into contract language
- Reevaluations to assess improvement against actions
- Manage supplier using defined Process
  - Ongoing Communication
  - Regular reviews and reevaluations
  - Required metrics




MOTOROLA
Satellite Communications Division

## Reevaluation Procedures

- Perform every 6 or 12 months (depending on size)
- Review supplier Issues with SATCOM program management
- Review improvement plan (if any) from initial evaluation
- Review Opportunities for Improvement from previous report
- Focus on these issues during reevaluation
- Notify supplier of focus of visit
- Conduct informally (mutual presentations and questions)
- Ask also how we may improve our management effort
- Send report to supplier and SATCOM program management

## Reevaluation Findings

- Personnel buildup is most common cause of schedule hit
- Quality initiatives occur where not previously found
- Process improvement in suppliers with poor previous focus
- Many suppliers have noted increased SEI maturity level
- Some suppliers come to benchmark us






## Lessons Learned - I

- Develop short schedule for COTS and small suppliers
- Need list of specific improvement areas, where necessary, before letting contract
- Small suppliers lack documented process but have an encultured process
- Size has little bearing on quality of process
- International suppliers have strong software capability








## Lessons Learned - II

- Follow up on requested improvements
- Use business analysis as an important indicator
- Look for shell organization that will use job-shoppers
- Conduct assessments for competing vendors in close succession
- Conduct assessments along with business analysts and domain experts






## Lessons Learned - III

- Domain expertise often requires us to use less mature supplier
- Work cooperatively with supplier to improve capability
- Process improvement easier to teach than domain expertise
- No one is as good as they say they are
- Assessment is first step -- must follow through to subsequent phases

# Assessment in Small Software Companies

Wolfgang B. Strigel
Software Productivity Centre

Vancouver, Canada
July 1995

## Abstract

The Capability Maturity Model was originally developed for large military and aerospace contractors. It represents an excellent framework of software process concepts and embodies some general principles which apply to companies of any size and product orientation. This, however, does not mean that the Software Engineering Institute (SEI) assessment can be used in small and medium sized enterprises (SME's) without some modifications. This paper describes three years of experience with a modified assessment technique for SME's. The paper also deals with the risks of modifying a recognized framework in an attempt to adapt it for the needs of more informal environments.

## Keywords

Software process assessment, process improvement, quality, productivity, SEI, CMM, small medium size companies.

## Biography

Wolfgang B. Strigel (strigel@spc.ca) is the founder and Managing Director of the Software Productivity Centre. Previously, he was Vice President of Engineering at MacDonald Dettwiler, a Canadian aerospace company, where he introduced a successful software engineering process improvement program using methodologies, standards and metrics.
Mr. Strigel has over 20 years of experience in software development and management. He received a B.Sc. (Munich, Germany) and M.Sc. (McGill University, Montreal, Canada) in Computer Science and a MBA (SFU, Vancouver, Canada). He is a member of ACM and senior member of IEEE. His particular interests are in the area of software process improvement. He is the author of several papers and a member of IEEE and ISO standards working groups.

## Introduction

This paper presents experience gathered in applying a modified version of the Software Engineering Institute's (SEI) Software Process Assessment (SPA) concept for SME's. It summarizes the results obtained at the Software Productivity Centre (SPC) from 10 assessments in this industry segment. The size of companies ranged from 50 to 200 total employees with a software development staff from 2 to 50 professionals. The companies included software product (shrink wrap) firms, system integrators, and embedded software firms.

The Software Productivity Centre is a non-profit technical resource centre located in Vancouver, Canada. It was founded in 1992 and has about 120 corporate members. Its main activities include software engineering training, assessments and consultation, and international sales of a family of process improvement products.

While gathering support material for this paper and reviewing some recent literature I started to become discouraged. There has been so much written about process improvement, and experience with the SEI Capability Maturity Model (CMM) that it is hard to believe there is any value in publishing yet another paper on this subject. My motivation sank even lower when I listened to Fred Brook's keynote address at the 17th International Conference on Software Engineering (April, 1995, Seattle, Wash.). He pointed out that it is ominous that his twenty year old book about a thirty year old story (The Mythical Man Month[1]) is still as relevant as the day it was published. The only argument that kept me from killing the idea was that I have always been a fan of experience reports which I find an absolutely necessary complement to theoretical papers. It is my hope that some pieces of information in this report will strike a chord somewhere in an organization and help to avoid retracing previously charted territory.

## Background

This paper assumes that the audience is reasonably familiar with the CMM concept[2], its Key Process Areas (KPA), Software Process Assessments (SPA), Software Process Evaluations (SPE), and the CMM Based Appraisal for Internal Process Improvement (CBA-IPI)[3].

Right from the start we, at the SPC, felt that process assessments would be of significant value to our member companies. The basic concept is simple, but powerful: an independent expert, who is an unbiased outsider to the company, takes a detailed look at how the company goes about software development and its associated process. As a result valuable feedback allows the assessed company to start or continue on its journey of continuous improvement.

## The Bandwagon Effect

Judging from industry feedback and from scanning popular software journals, a considerable momentum is building in the field of process improvement. After years of preaching the gospel it seems that the industry at large is finally taking note of the importance of process improvement, whether it is influenced by the SEI, ISO 9000 or other motivations. The success of SPC activities which are primarily focusing on process improvement issues is another indication.
Initiatives such as the SEI or the Software Productivity Consortium were started in the 80's primarily to deal with major problems in the development of large aerospace or military systems. Over the last few years, however, other software industries have seen significant increase in the size and complexity of their software development projects and are now facing similar problems. As a result we observe some sort of bandwagon effect (Figure 1) where more and more software product companies are concerned about the development process and not only the technological issues. Our focus at the SPC is to enable companies to move from level 1 maturity to higher levels of process maturity which correspond to levels two and three but do not necessarily include all requirements for the corresponding SEI ranking. Our software process assessment is one vehicle we use to increase the maturity level of software companies.

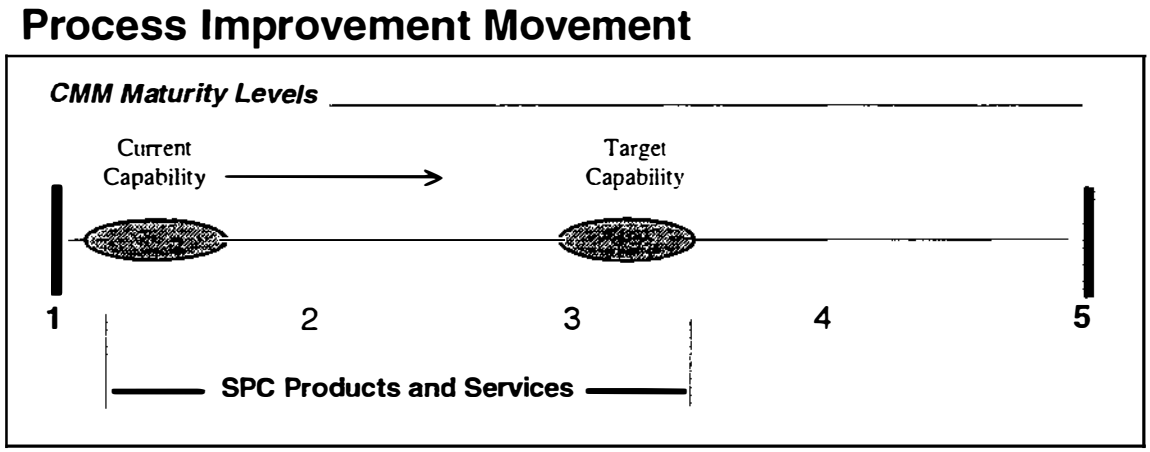



Figure 1: SPC Support to Increase Process Maturity

## The SPC Assessment Approach

Our objective for conducting assessments is to:

1. Diagnose the current state of the development process using some proven frame
2. Provide practical recommendations for improvement and prioritize them by order of impact divided by cost (or "bang for the buck").

We were aware of the SEI activities and we evaluated the CMM and maturity questionnaire in light of the needs of small and medium sized software companies[4]. In SEI terminology we needed an *assessment* instrument not just an *evaluation* (an assessment involves company staff and results in recommendations while an evaluation is more like an audit, conducted by external assessors with the goal to derive a capability level). Our focus was to generate recommendations

for improvement. We felt that this was more important to our members than to offer a ranking such as the SEI maturity scale (1-5). Our target audience was almost exclusively dealing in commercial markets and several changes to the maturity questionnaire (MQ) were needed. However, our fundamental premise in approaching this issue was to stay as close as possible to a proven software process standard instead of developing yet another evaluation baseline. The CMM and MQ provided us with this framework. Any changes to the original MQ (and later the CBA-IPI maturity questionnaire, V1.1.0) were kept to a minimum and although we modified the assessment process we always stayed as close as possible to the original SEI concept.

Our initial questionnaire was based on SEI's original MQ. First, we considered the need to cover all 5 levels of the maturity model. This exceeded our requirements given that most of the companies we dealt with were at SEI level 1. In our opinion level 3 was as much as any of these companies would aspire to achieve, at least within the near future.
In reducing the questionnaire we did not strictly cut out all questions related to levels 4 and 5. Our target was to not exceed 70 questions in total to keep an interview session at a maximum of 1.5 hours, which is the maximum attention span we could expect. This gave us an average of 2 minutes per question since we always counted on some questions which are not applicable, and a few which can be answered very quickly. We eliminated some questions (about 8%) from levels 1 to 3 which we felt were repetitive. On the other hand there were some questions related to levels 4 and 5 which we included.
A later revision was based on the MQ Version 1.1.0 which SEI completed in 1994. We changed the wording of the questionnaire to make it more understandable for commercial firms. We also removed the assumption of highly structured organizations with formal QA departments (most of our assessments encountered a very simple and flat organizational structure).

Examining the assessment process as prescribed by the SEI it was obvious that it exceeded the needs and resources of our target audience. We determined that for an average company with 20 software engineers the whole assessment should not cost more than $5,000 and take no more than a total of 50 staff hours of the assessed company's staff for a total effective cost of less than $10,000 or an investment of $500 per software developer. Our experience has shown that higher costs can be a major impediment for SME's. The SEI process with its group sessions, training, focus groups etc. would have taken considerably more. Unfortunately, we were not able to include company staff in the assessment team and to train them in assessment techniques since this proved too costly and disruptive for SME's.

We also felt that the assessment approach is lacking one major aspect. It seems to assume that all companies have the same needs, regardless of their size, their business objectives or the environment in which they operate (technology and market). In reality, company objectives and strategies are a function of the technology and business environment in which they operate (see Figure 2). To evaluate a company without consideration of this external context can lead to recommendations which do not meet the most critical company needs. Although the SEI encourages assessors to use some degree of their own judgment in the way they approach an assessment there was no guidance which would consider above mentioned factors. In our opinion this was a significant drawback. Given the diversity of company types and sizes we could not apply the same standard to all companies and expect relevant results.

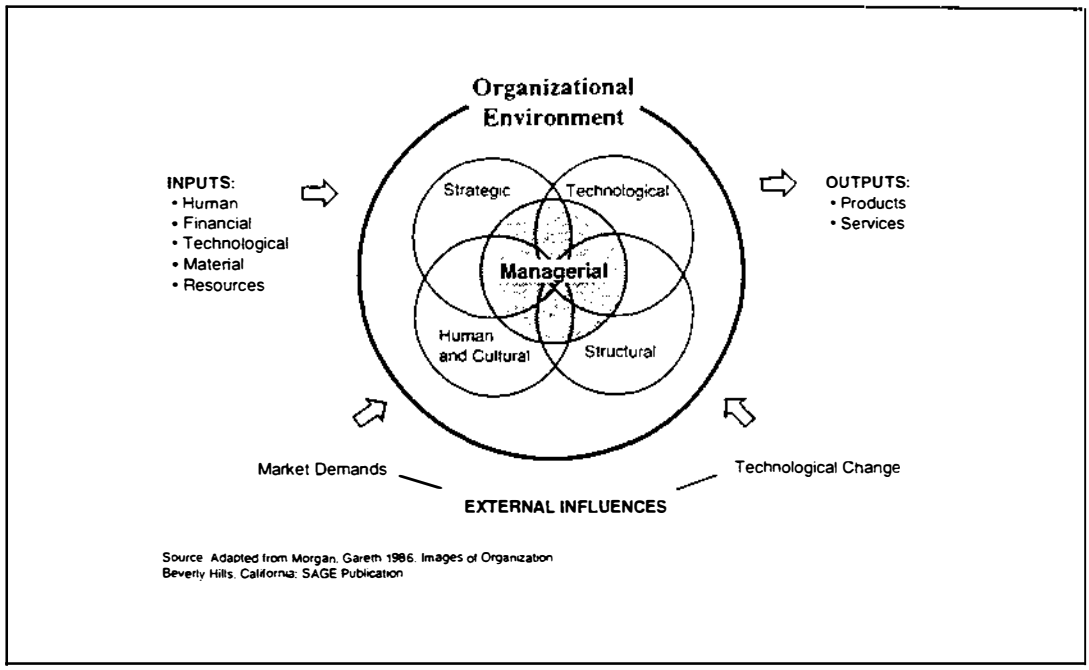



Figure 2: Internal and External Company Influences

Ultimately, the direction of each company and its operating strategy in this environment is defined by senior management. We therefore added a step to the assessment process which is to interview the President, the VP of marketing and the VP of development (or equivalent positions) to determine the priorities and key issues as seen by these senior executives. The outcome of these sessions (each of approximately one hour) served as input to the remainder of the assessment. It did not alter the questionnaire but it provided the context for our analysis and recommendations. These management meetings also serve to establish a firm management commitment for the assessment process and for follow up actions to implement the recommendations.

Examples of corporate priorities obtained in this way are:

- time to market,
- product quality,
- product usability, and
- technology leadership.

It stands to reason that a company which sees its highest priority as being the first to release a new product (time to market) and which is willing to trade off quality (to some extent) should focus on different process areas than a company which puts highest emphasis on technological leadership.

## SPC Assessment Steps

As a result of the above considerations a typical SPC assessment starts with three management interviews. These are followed by a one hour kickoff meeting with as many software developers as practical. The objective is to explain why the company decided to undergo an assessment and to highlight some of the key issues mentioned during the management interviews. We also give an outlook on the planned assessment activities and results. This session is useful to introduce the assessors and underline the commitment by senior management to improving the current process. In addition we mention that the success of the assessment process depends on the collaboration between assessors and the development team. Finally we stress that all discussions and input will be treated with strict confidentiality.
The next step is to conduct about 3 to 5 interview sessions with members of the development team. The interviewees are chosen carefully to represent a cross section of team leaders, unofficial leaders (i.e. key individuals who are respected for their expertise and exercise unofficial influence on the team) and other team members without specific leadership roles. We also schedule one or two group sessions with about 5 software engineers each. These sessions are intended to flush out issues or comments which are often missed in the questionnaire driven interviews. These meetings also allow us to explore in more depth some questions raised during the interviews as well as specific issues related to the corporate objectives. In addition, these sessions help to start improvement actions through the group dynamics of developers discussing their issues in a neutral and confidential environment.

After analyzing the findings we compile a draft report which is discussed with management to ensure that their key issues are addressed. Finally, we hold a debriefing session with the whole team (same as the kickoff meeting) to present our findings and recommendations. At least half of this meeting is reserved for discussion. The dynamics of this meeting are very important. It allows to foster a group consensus on the need for improvement, reinforces the relevance of the recommendations and fosters support for the ensuing process improvement activities.

Our final report is very succinct and to the point. Unnecessary verbiage and generalities are avoided. Key findings and recommendations are highlighted in bold typeface to make sure that the reader has no doubt about the message we are conveying. Earlier experience has shown that senior management sometimes only skims these reports. We therefore use at times very blunt statements to make sure the report gets the desired attention. To make the recommendations practical in consideration of economic realities we summarize the top 5 recommendations which in our opinion have the highest impact in meeting corporate objectives.

## Summary of SPC Assessment Results

To discuss the results of our assessments we could present a statistical analysis of the questionnaire responses. However the sample size is not sufficient to produce a meaningful result. More importantly we feel that the report summaries which identify weaknesses and give recommendations for improvements are more relevant since we use the questionnaire only as a framework for our assessments. The summary of 10 assessments, as seen in Table 1, therefore focuses on the most frequently observed problem areas which seem to be symptomatic for small

and medium size organizations. Many of the issues listed in Table 1 are addressed by the CMM key practices and the majority of them map back to level 2 and 3 KPA's of the CMM.

Each of the 10 assessments in Table 1 is labeled alphabetically. The presence of a letter in the matrix indicates that the related issue was identified as a problem in this assessment. The "Total" column indicates the number of assessments in which a specific issue was identified. The issues were then ranked by the frequency of occurrence.

| Issues | Assessments | | | | | | | | | | Total | Rank |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Requirements Management | a | b | c | d | e | f | g | h | i | j | 10 | 1 |
| Design Documentation | a | b | c | d | e | f | g | h | i | j | 10 | 1 |
| Metrics | a | b | c | d | e | f | g | h | i | j | 10 | 1 |
| Engineering Process | a | b | c | d | e | f | g | h | | j | 9 | 2 |
| Tools | a | b | c | d | e | f | | h | i | j | 9 | 2 |
| Project Control | a | b | c | d | e | f | | h | | j | 8 | 3 |
| Organizational Structure | a | b | | | e | | | h | | j | 5 | 4 |
| Testing | a | b | c | | | f | | | i | | 5 | 4 |
| Quality Assurance | a | | | d | e | | | | | j | 4 | 5 |
| Product Management | a | | | | e | | | | | j | 3 | 6 |
| Technology Management | | | | | e | | | | | j | 2 | 7 |

**Table 1: Assessment Results by Frequency of Recommendation**

Assessments *a*, *b*, and *c* represent assessments of the same company in three consecutive years. Assessments *h* and *i* identify another company which was reassessed after two years. In the first case only slight improvements were achieved every year. In the second case the company started from a position of better process understanding and achieved major advances based on recommendations from the first assessment. The remaining assessments represent companies which were assessed once.

## SPC Assessment Results by Priority

Table 1 showed our results ordered by their frequency of occurrence. However, this in itself is not an indicator of the financial and quality impact on the organization. Therefore, the second objective was to identify issues which offered significant leverage for obtaining quick and lasting benefits. Table 2 summarizes the five highest priority issues which meet the second objective. The priorities are a result of our observations during three years of process improvement consultation in SME's. Each is subsequently described in more detail.

| Priority | Issue |
|---|---|
| 1 | Requirements Management |
| 2 | Design Documentation |
| 3 | Engineering Process |
| 4 | Project Control |
| 5 | Organizational Structure |

**Table 2: Issues Prioritized by Potential Pay Back**

**Issue 1:** The most damaging problem was the omission to manage requirements. Many of the problems expressed by senior management such as low estimates, lack of schedule control and missed functionality are rooted in this area. In many cases the requirements were not documented at all or described insufficiently. The lack of managing the requirements during the development led to uncontrolled expansion of features. The scope of the project expanded and soon the initial estimates became meaningless.

**Issue 2:** Insufficient design documentation was also identified as a high impact area. Similar to the previous issue the absence of a defined approach to design documentation led to uncontrolled changes in functionality with the resulting schedule impacts. In addition it exposed the company to significant risk exposure because all design knowledge resided only in the heads of a few engineers. People could not easily be reassigned to other projects and if any of the designers left the company it was impossible to transfer the design knowledge.

**Issue 3:** Without a defined engineering process the development organization has no framework on which to base a controlled process. At a minimum there needs to be some document which describes the basic steps and procedures for software development. The description does not need to be very voluminous but it should contain a development life cycle concept and some rules about the transition between life cycle phases. Reviews must be a defined part of the process to enable these transitions to take place.

**Issue 4:** Once the process and a documentation scheme are defined, project control can take place. We found that a common problem in SME's is that team leaders and project managers are appointed but no training in project management or people management skills is offered. Much effort is wasted when these newly appointed managers experiment with control mechanisms and refine their own flavour of project management through trial and error.

**Issue 5:** Our final candidate in the list of the top five issues is organizational structure. Fifty percent of our assessments indicated a problem in this area. Most of these organizations experienced rapid growth, some doubling their development staff every year. They quickly reach a point at which the original technology managers may be more effective to continue focusing on

product technology. A senior manager with a strong software engineering process background is required to oversee a growing number of projects. The main function of this position is to foster the increasing formalism of the development process and coordinate the communication of the development department with other departments in the organization. We discovered many instances where the wrong people were leading these groups. Their communication and people skills were no match for the challenges of the job. In a more general sense, a company needs the infrastructure to enable effective management on continuous improvement and change.

The selection of the above top five issues is not without controversy. In many cases the prioritization depends on the size of the organization, the complexity of each project and other factors which are specific to each individual case. The fundamental issue is that it is relatively easy to conduct an evaluation using a framework such as the CMM but it requires a lot of experience to translate the findings into recommendations which optimize the benefits in light of organizational priorities.

## Assessment Challenges

There are several problems with SEI-type assessment techniques. For instance we often feel that after one or two interview sessions any further interviews do not yield much additional information. We then take the liberty to deviate from the questionnaire to home in on specific problem areas. These sessions are still inspired by the overall assessment framework and generate a lot of handwritten notes. This can easily lead to an assessment which "becomes a loosely directed, intuitive exploration"[5]. At this point the value foundation of the assessment shifts from the assessment framework (i.e. the CMM) to become a function of the experience and intuition of the assessor. This is acceptable if the assessor has suitable qualifications. As is frequently the case, the answer seems to lie in a compromise. We feel it is acceptable to use the formal assessment framework as a guide and to cater to the individual needs of each target company by adding our own judgment. This enables us to consider the particular objectives of each organization and to derive recommendations which meet their particular needs.

For very small organizations the above concern is driven to its extreme. The questionnaire just becomes too overwhelming and the concepts of process improvement have to be approached through simple educational steps. In these cases the questionnaire would tend to become an instructional tool which identifies all the good things the organization could or should do. But the questionnaire was not intended as an educational tool and some more fundamental techniques of awareness building have to be used.

A dilemma we frequently face is the question about how much formalism should be introduced or is justified in a small organization. Standard questions which start with the menacing phrase "is there a defined mechanism to..." just don't make sense in some cases. How much process documentation should exist in small organizations, and how much formalism should be applied to control compliance? Our approach in making recommendations is again very much directed by our judgment. In some organizations a 10 page document describing the development process is

sufficient and probably all one can expect as a first step. As a process is beginning to take shape, progressively more rigorous process elements are introduced.

Finally we would like to dwell on the big issue of management commitment. It is by far the biggest challenge to obtain *true* management commitment. Our approach is again directed by the particular circumstances. In all cases we need to find the one issue that gets the president's attention. Preaching the gospel about the benefits of quality or how wonderful a 10% productivity increase would be does not work in most cases. Initially we thought it should, but after years of trying we have learned that lesson. What we are now looking for is the *sore spot*. Examples are a recent configuration control disaster (shipped the wrong version to 1000 customers), a schedule slip which made the firm miss a major trade show, etc. If we have a convincing argument that assessments and the resulting improvements can effectively deal with these issues we have a good chance for commitment which lasts for some time (it will need constant reinforcement). A good example of short lived (or partial) commitment is in the results of assessments *a* through *c* where the same company asked for our help in three consecutive years to deal with some of the above mentioned disasters. However, as soon as there was an emergency, the president rearranged priorities, overriding dearly acquired process improvements in favour of serving the latest market pull. The result was only marginal process improvements over the three year period. We have found some ammunition but certainly no silver bullet to solve this issue of management commitment.

## Recommendations and Outlook

Evaluations are generally the easy part, recommendations are much more difficult. Our focus is to identify the 20% effort that achieve 80% payback. We could possibly develop a formula or a structured approach to achieve this. Barry Boehm did it in Cocomo[6] where many input components from people to tools are given weights to calculate estimated project cost. Many of these factors are well known, for example, people are a key factor. But this people factor was not a problem in the SME's we assessed. The reason is probably that if a company starts out with poor staff they simply don't survive for very long. Table 2 gave the top five priorities to achieve the 80/20 goal according to our experience. But each company is different and a rigid formula for process improvement priorities can cause problems if applied without the backup of years of experience and a healthy dose of professional intuition. This includes sensitivity to the company's culture and the limitations the culture may impose on the introduction of change.

## Tool Support

Continuous monitoring of the process is a necessary ingredient to assure lasting success of an initial assessment. It requires a champion who makes sure the momentum does not fade away and it requires the instrument for effective re-assessments. The latter could be an external assessor but many companies resist the repeated use of consultants. During our assessments we found an almost complete absence of tool usage to support continuous process improvement. Effective tools can range from simple checklists or templates to systems which assist in the monitoring of

progress in the improvement process. To facilitate the continuing improvement process the SPC has developed several tools which can assist the champion in his work. The first is a metrics program and metrics repository which helps the firm in tracking its progress. The second is a computerized assessment instrument which guides the assessor through our modified questionnaire. This tool offers checklists, action plans and practical advice that allows the company to do its own repeated reassessments. The assessment results are shown as bar chart profiles and improvement trends between successive assessments can be charted. We also developed a large number of soft copy documentation templates. Our experience has shown that even such a simple tool can offer significant help. There is an increasing number of process related tools on the market and many are quite affordable for SME's.

## Summary

The SEI CMM is an excellent framework for process evaluations. But working with an SME requires a flexible approach and adaptation to particular needs is required. The key question remains: how far to deviate before relevance of the process is lost. We have no clear cut answer and we are not convinced that a simple solution exists.

It is fairly certain that a standard approach to assessments works in large organizations (over 100 developers) as long as it is based on a proven framework. Results from SEI assessments and other derivatives such as BOOTSTRAP[7] have shown this. As the size of the development organization decreases the assessment methodology becomes increasingly informal. However, for large and small companies, there is no easy formula to offer improvement advice which produces the maximum benefit for a minimum investment (or in financial terms the approach which maximizes the return on investment).

Our observations indicate that key practices for SEI level two and three are most relevant for SME's. Our modified questionnaire serves as a starting point to gain an initial understanding of a company's strengths and weaknesses. However, the results obtained from the interviews must be considered in light of company objectives and priorities to derive recommendations that are relevant to the company.

## Acknowledgments

I would like the acknowledge the very valuable input and review suggestions I received from my colleague Alice Kloosterboer at the Software Productivity Centre.

## References


1) Brooks, Frederick, P.; "The Mythical Man-Month", Addison-Wesley, 1975

2) Paulk, Mark, C. et al; "Key Practices of the Capability Maturity Model V1.1", SEI, February 1993

3) CBA Project Team, "CBA-IPI Method Description Document - V1.0", CMU/SEI-93-TR-25

4) Strigel, Wolfgang, B.; "Industry-Wide Software Process Improvements, PNSQC, 1993, Portland, OR

5) Kitson, David, H., Humphrey, Watts, S.; "The Role of Assessments in Software Process Improvement", p. 17, CMU/SEI-89-TR-3, Dec. 1989

6) Boehm, Barry, W.; "Software Engineering Economics", Prentice Hall, 1981

7) Kuvaja, Pasi, et al; "Software Process Assessment & Improvement - The BOOTSTRAP Approach, Blackwell Publishers, Oxford, UK, 1994

# A Useful Framework for Adopting Software Metrics

Barbara Zimmer
Hewlett-Packard Company
1501 Page Mill Rd 5MR
Palo Alto, CA 94304
zimmer@ce.hp.com

## Abstract

Hewlett-Packard has received considerable attention for its work in software measurement. Pockets of success are evident in many parts of the company and there is consistent widespread use of a limited set of software metrics, spurred in part by HP's now ended Software 10X improvement program. A relatively recent effort called the Software Initiative (SWI) has taken a different approach to process improvement in general, and metrics adoption in particular. Composed of engineering and management experts, the SWI consulting team partners with HP divisions to optimize their investment in software development capability.

SWI's outcome-directed approach impacts not only how we consult, but which products and services are developed and delivered. In particular, we have found it useful to focus on five specific areas in considering how best to apply the Goal-Question-Metric paradigm. This paper will discuss these focus areas, HP's experience with them and how it has effected the scope of current software metrics efforts at HP.

## Biographical Sketch

Barbara Zimmer is a consulting engineer with Hewlett-Packard's Software Initiative (SWI) program. In 12 years at HP, she has worked in Finance, Information Systems, and now R&D. She is currently consulting with HP R&D customers in the areas of software measurement and internal communications as well as working on researching, developing, and packaging products and services that support the SWI's consulting efforts in such areas as software reuse, failure analysis, and evolutionary product development. She has twin 4 year-old daughters who have taught her the importance of knowing and articulating where you are and where you want to be.

## Introduction

Hewlett-Packard has received considerable attention for its work in software measurement. Much of this attention has been due to the publication of two books by Bob Grady and Debbie Caswell, [1,2] as well as industry coverage of the company's Software 10X Program. The current software metrics focus in HP is on supporting the needs of individual HP businesses. The Software Initiative (SWI) exemplifies one current approach to supporting the varied software development needs of HP's businesses.

SWI is a centralized organization offering consulting services to software development groups throughout HP. Launched in 1991, SWI has learned much about how to affect software development process improvements in general and about implementing software metrics, in particular. This exploration of SWI's experiences in enabling change at HP examines the background for SWI's creation, the group's organization and approach, and how this has influenced our thinking about implementing software metrics.

## Background

It is helpful to start with a brief look at HP's 10X software improvement goals set in 1986. While software metrics have been applied with varying degrees of rigor and impact throughout the company, perhaps the biggest impetus for raised awareness of metrics in general was the Software 10X program. This program set corporate goals of improving defect density and number of open serious and critical known problems by a factor of 10 in five years. Several HP divisions did achieve 10X improvement in defect density and HP as a whole succeeded in reaching a 6X improvement.

Important to the success of the 10X Program were the discoveries made in the process. On one level, we were able to learn from the individual HP organizations which achieved the 10X goal.• All of the divisions who reached the 10X improvement goal committed to release only well-tested, low defect products and had strategies that emphasized an organization-wide commitment to quality. The divisions used a variety of methods to improve quality, including design techniques, defect detection and defect prevention.

On another level, the company learned that the 10X goals themselves were reasonable for most but not for all of HP. The size, organizational structure, culture, and wide ranging business goals of HP's many businesses makes the feasibility and even desirability of any company-wide set of software standards problematic. Further, the enormous and growing increase in the amount of code created, due to both increased rate of product introductions and volume of code per product (see also [3]) meant the actual improvement in open serious and critical known problems called for was

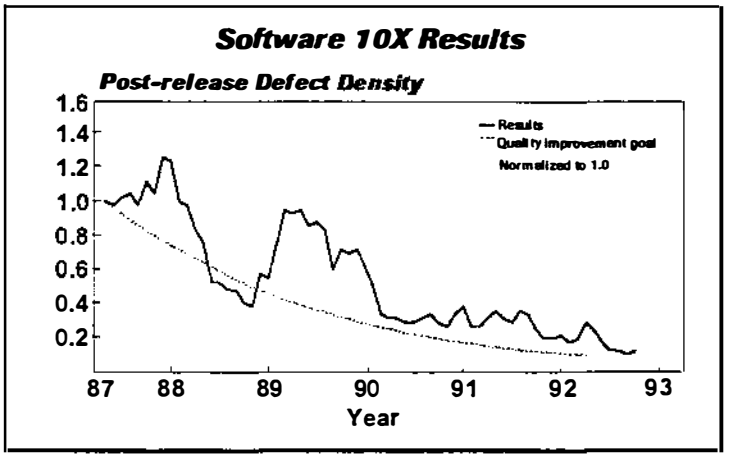



*Figure 1*

• HP is organized into Business Sectors and then Groups which include multiple divisions. Each division is relatively autonomous and includes all normal business functions (marketing, manufacturing, quality, R&D ...). A typical R&D lab consists of three or four sections, each of which contains three or four project teams of three to ten engineers.

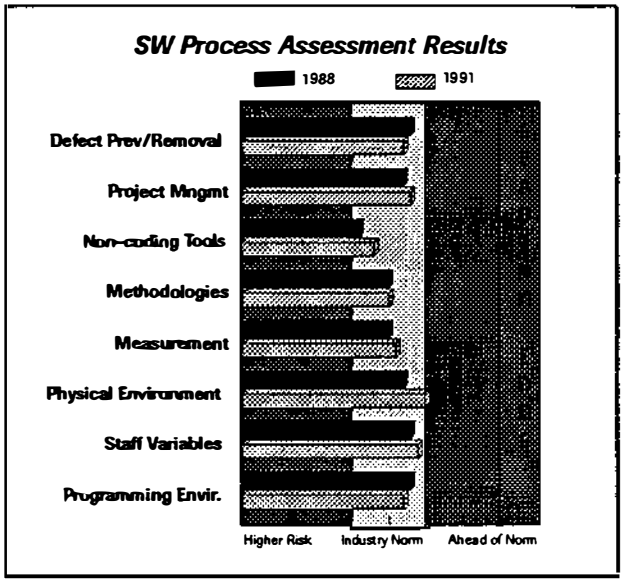



*Figure 2*

effectively much more than 10X in order to reach the 10X program goal.

HP management also recognized that the company's underlying software processes weren't good enough to sustain changing business needs and there was not an adequate infrastructure in place to support desired improvements. This recognition was further fueled by several years of software process assessment data (using an assessment method that predated the now popular SEI methods and summarized in Figure 2) which indicated that while there were some isolated success stories, the company as a whole was not gaining ground on software development processes.

One response to this information was the creation of the Software Initiative (SWI) to provide leadership in driving new software development methods and technologies through process change. Composed of engineering and management experts, the SWI consulting team partners with HP divisions to optimize their investment in software development capability. The SWI team works with multiple levels of group and division management to understand the entity business goals and explore ways of gaining competitive advantage. These goals, along with other challenges or obstacles that need to be considered, drive the creation of specific improvement plans. All SWI customers are internal HP business entities. This basic operating model is illustrated in Figure 3.

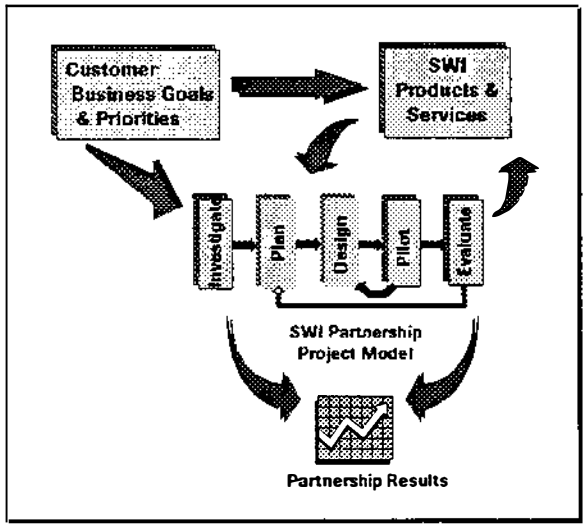



*Figure 3*

Initially, the SWI undertook its charter by organizing itself around six technical areas which it had determined had the highest payback on process improvement investment. Figure 4 lists those focus areas as well as the sources of information which contributed to their determination and prioritization. It soon became apparent that, while this could effect some short term improvements, it was not leading to the kind of long term change that was desired. The group gradually evolved its orientation from purely tactical to one which emphasized the strategic element. It explicitly recognized that software engineering expertise must be coupled with enough engineering management expertise to make the desired changes take hold.

Further, the effectiveness of the *delivery* of this expertise by SWI team members was perhaps the most critical success factor for affecting lasting change. This orientation shift was consequently accompanied by gradually building a strong suite of consulting skills on the foundation of technical and management expertise that SWI team members had brought to the Initiative.

## The Software Initiative Approach

SWI's mission is to deliver software knowledge and expertise in key software competency areas that enable HP businesses to gain competitive advantage. To achieve this, SWI consultants' work is driven by customer-directed outcomes. In practice, determining what these outcomes are is often in itself a significant contribution to the customer organization and is the first step in any SWI consulting engagement.

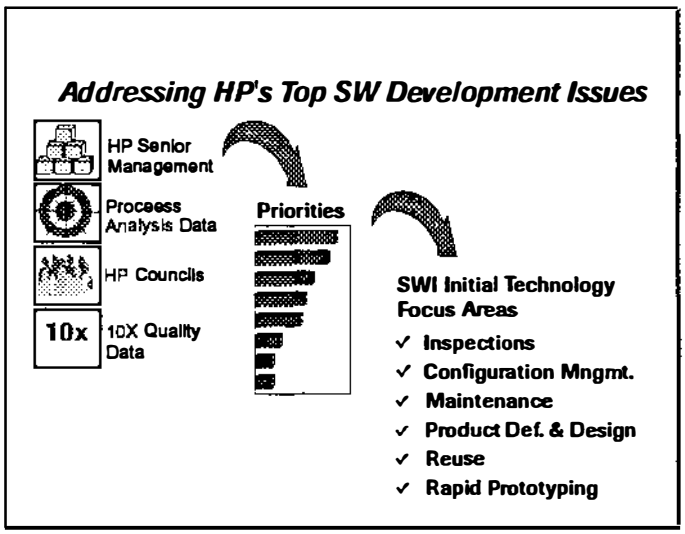



*Figure 4*

SWI has defined its customer interface with an Account Management System. The SWI is essentially one large resource pool from which teams are formed "just-in-time" to address specific customer needs. The customer works directly with their SWI Account Manager who then draws an appropriate team or individual from the SWI pool based on his or her assessment of the customer need, the individual's specific expertise, and availability. Teams are together only as long as they are actively meeting customer needs and individuals are members of many teams at any given time. The Account Manager provides the constancy in the relationships and processes to minimize customer confusion and optimize the use of limited resources.

This structure allows great flexibility and responsiveness to the widely varying needs of customers. It also yields greater efficiency in working with customers whose real-time product development schedules constantly impact the time available for process efforts. If one project gets put temporarily on hold, there are plenty of other projects to take up the slack. Demand for SWI services has exceeded our delivery bandwidth so it has not been necessary to "market" our services.

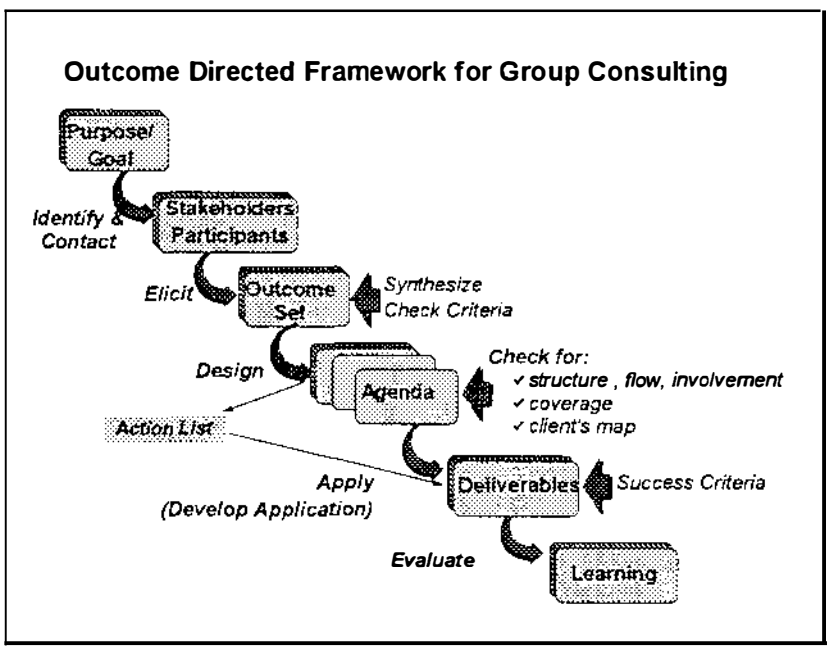



*Figure 5*

## Applying an Outcome-Directed Approach to Software Metrics

This outcome-directed approach has impacted not only how SWI delivers consulting, but which products and services are developed and delivered. In the area of measurement, it has led to some rethinking of how we apply the Goal-Question-Metric (GQM) paradigm. Basili's initial presentation of the GQM paradigm[4,5] has heavily influenced the evolution of a software metrics course that was revised in 1990 and is currently taught inside HP.[6]

The process of establishing a workable set of goals, asking questions to clarify, constrain, and

focus these goals, and then identifying metrics that will provide answers to the questions has provided a useful framework for SWI's efforts in metrics adoption at HP. Reed has reported on metrics work which has similarly seen benefit from this goal and strategic planning orientation, as well as from an emphasis on communication.[7] Indeed, putting a high priority on the effective communication of any information gathered along with determining the level and pace at which one can reasonably expect to implement a metrics program, have proven to be highly significant considerations in determining how to apply GQM.

At HP, we have consequently found it useful to focus on five specific areas in considering how best to adapt the paradigm. These areas are

- organizational chunk (what part of an organization to work with)
- organizational readiness
- specific process areas of concern
- pacing the adoption process
- communicating the results.

The first three of these are closely linked and somewhat interdependent. The issue of how best to support the metrics adoption process underlies all five areas.

As noted earlier, HP's current metrics activities are not driven on a company-wide basis. Discussion of the five focus areas will help explain why extending HP's current metrics practices involve fairly small portions of HP organizations.

*Organizational Chunk*

Organizational chunk includes determining what is the appropriate size or part of an organization with which to work. SWI's customer engagements usually involve contracting at the division R&D lab level for work to be done within a segment of a lab, often a section consisting of several project teams. In planning a metrics project for the section, we would first determine the desired outcomes for the metrics project itself.

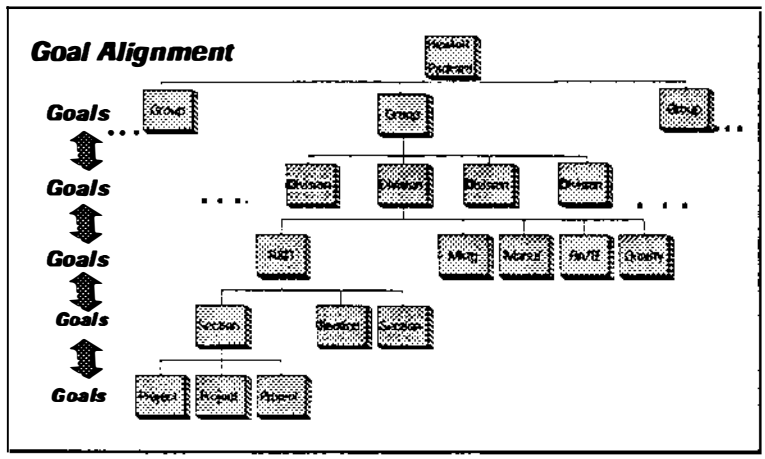



*Figure 6*

To apply the GQM paradigm, we need to assess which goals to address first - the lab's, the section's or the projects within the section. Since the lab is most likely to have already established business goals and is not likely to change them in the short term, we can usually accept the lab goals as given. We then work with the project teams and section to claify goals and make sure they are aligned with the lab goals.

Several questions can be asked to help determine the appropriate organizational chunk with which to undertake the metrics adoption work. First, *what is the largest natural grouping in the organization that shares a common environment and business goals?* In HP R&D labs, there is often one software or firmware section which delivers software or firmware to product teams in other lab sections. This section would probably be the "largest natural grouping."

*What higher level organization does this group need to align with?* In HP, that will usually be the lab and maybe the division. In some cases the Group level goals are strongly coupled to a particular section's results.

*What component groups have a stake in the outcomes and successful implementation of the project?* This is usually the project teams within the section. Aligning project goals may not seem to be critical to the project. However, like many organizational change efforts, it is the process of setting and aligning the goals rather than the goals themselves which contribute the most toward group understanding and buy-in in the long run. This can be done through a combination of individual interviews and group meetings.

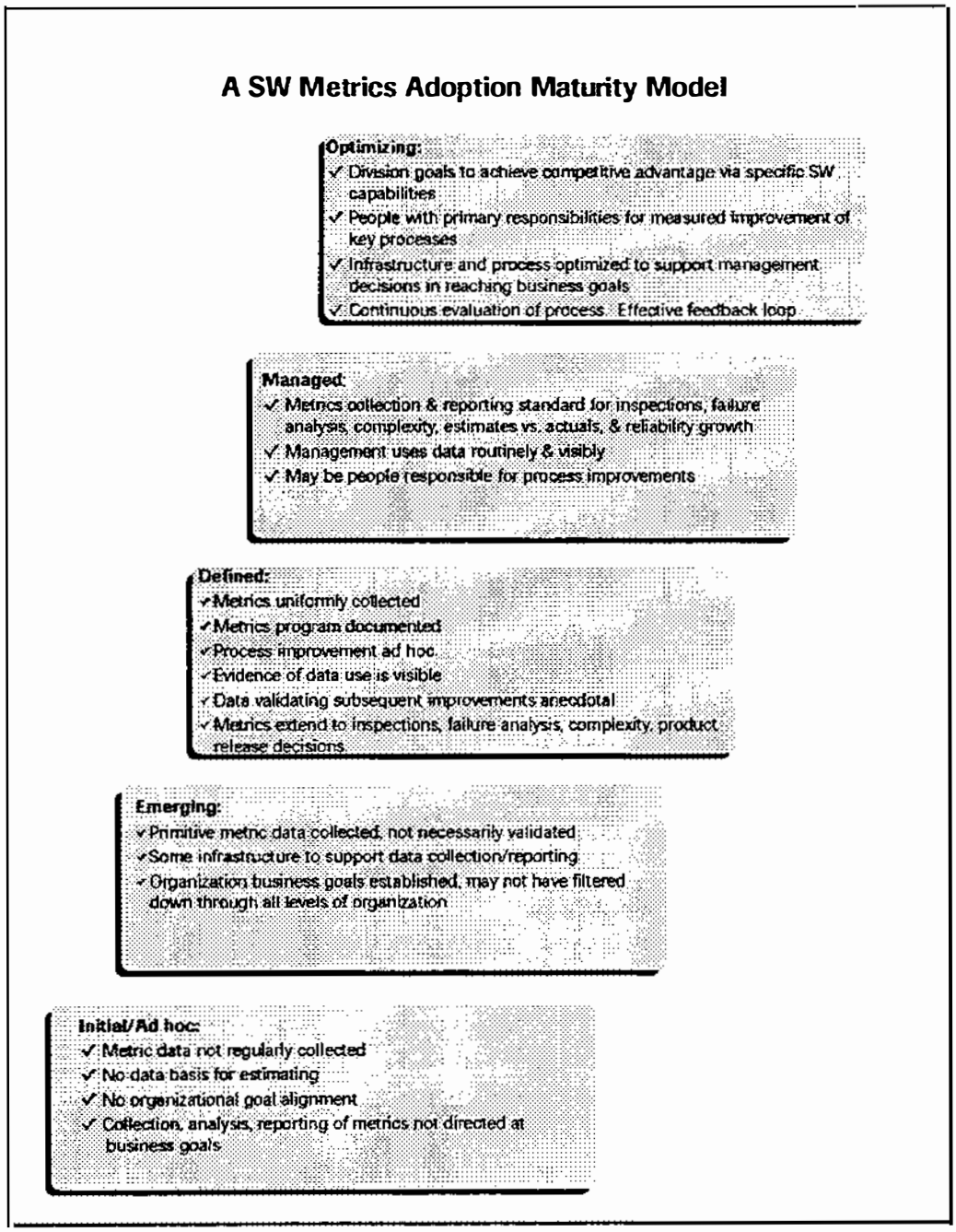



*Figure 7*

*Organizational Readiness*

The right solution for a particular group must consider organizational readiness, which here includes current technological and software development capabilities. SWI has adapted SEI's Capability Maturity Model to a general framework for evaluating maturity in a given process area. The framework is gradually being applied and validated with a number of software development processes, including metrics.

We are just beginning to validate this model's usefulness in determining the right set of metrics with which to begin (including the number and scope of the metrics) and a feasible implementation plan. It can also help to determine the appropriate organizational chunk as discussed above. For example if one project within a section was at a higher level on this model than other section projects, it might be appropriate to start to pilot metrics within this project rather than at the section level. Conversely, widening the gap between parts of an organization can hinder later efforts because of communication gaps and perceived inequalities.

*Specific process area of concern*

SWI has found that a key to enabling progress toward business goals is the application of metrics to the specific process areas associated with each goal. For example, a reuse organization will not move forward without evaluating specific reuse issues relative to business goals. Current focus on a particular process area may elevate the need and perceived urgency to be able to measure progress. Leveraging this attention can help jump start an ultimately more broad-based measurement program.

In some cases, concern about a new or challenging process may be the impetus for implementing metrics. For example, an organization adopting object-oriented methods will ask questions that will help them assess the value of the methods. Additionally, they may want to measure aspects of the learning curve that will help them predict the cost of adding new engineers.

*Pacing the metrics adoption process*

One of the SWI's key learnings in the first years of working with its many diverse division partners, was the importance of pacing: pacing the relationship, pacing each activity, and pacing the implementation of any planned improvements.

When practiced as a full enumeration of goals, questions and metrics, the GQM approach can easily overwhelm an organization before they even get to the implementation stage. The result can be lost momentum for metrics adoption or circumventing the GQM process entirely in favor of metrics that may contribute nothing to managing toward their business goals.

Beginning with the assumption that implementing metrics will be an evolutionary process, you can start pruning the GQM activities right from the start. Goals should be stated as well-formed, strategic objectives (see Figure 8) . The next step is to identify which goals require metrics in order to manage progress. Then decide, of those which need metric support, which are most critical?

With the critical goal(s) identified, you can select a small set of key questions whose answers enable the organization to assess progress. Energy is now focused on determining the few metrics most useful for reaching critical goals. Information regarding organizational chunk,

***The Goal Statement should be:***

- ✓ Specific and generally understandable
- ✓ Within area of responsibility
- ✓ Measurable
- ✓ Fixed time period
- ✓ Stretch

*Figure 8*

organizational readiness, and specific processes will contribute to arriving at feasible, useful and critical metrics to form the initial implementation set.

Taking an evolutionary approach to metrics adoption runs directly counter to our cultural desire to achieve major breakthroughs. This is yet another reason why constant and consistent communication is critical to sustaining a sense of forward motion.

*Communicating results*

The lasting success of software metrics ultimately depends on effective communication and visible use of the data collected. Unless people from all levels of the organization understand what the data are communicating and see for themselves how the information is being used to make management decisions, the metrics effort will likely be abandoned in short order.

For this reason, emphasis needs to be placed early in the project on how to present results. This includes how collected data will be synthesized and interpreted, how it will be visually/physically presented (including the appropriate medium to use), how the use of the data will be made visible to members of the organization, and what decisions will be made. Some guidelines for a communication plan are offered in Figure 9.

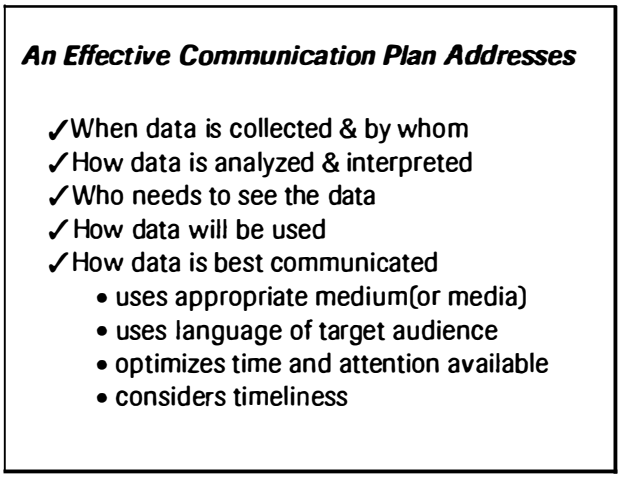



*Figure 9*

Leaving this activity as an afterthought to the metrics implementation can indeed be fatal to the project. If the appropriate managers are not getting the information they need to make good decisions, then the metrics collection will have been wasted effort. Further, if participants don't *see* their data being used effectively, they will *perceive* their effort to be wasted.

## Illustrating the SWI approach

Two SWI customer cases help to illustrate how this framework can shape the adoption of software metrics. In the first case which I'll call ABC, a metrics adoption project evolved out of a partnership that initially began with a focus on software reuse. The ABC lab emerged from a major reorganization designed to better support software reuse. ABC develops reusable core firmware and software for products built by several divisions on the same site. SWI was working with many different chunks of the organization before considering a metrics project.

The right organizational chunk was first proposed to be the entire lab. However the great disparity in organizational readiness between the software and firmware sections argued strongly for addressing just two firmware sections' needs first. The first step was to set lab-wide goals which would of course ultimately be used for aligning all three sections. We then individually interviewed each of the managers of the eight firmware projects in order to ascertain their project goals. Most of the project managers had not consciously considered their project goals in

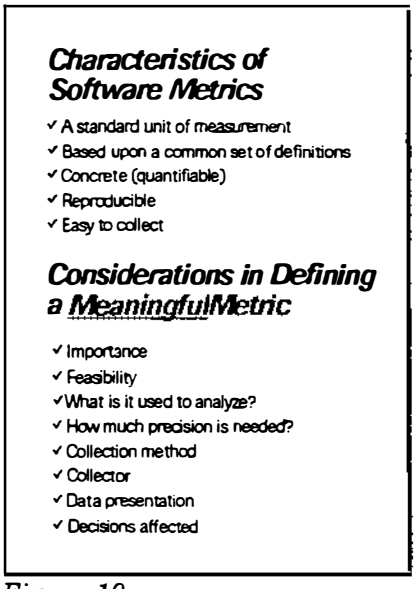



*Figure 10*

relation to lab or division goals. We synthesized the information from these interviews, presented it back to the group of section and project managers, and went through an exercise of writing at least one well-formed objective for each project. Together, we also took the section managers' goals and agreed on a well-formed set for each section.

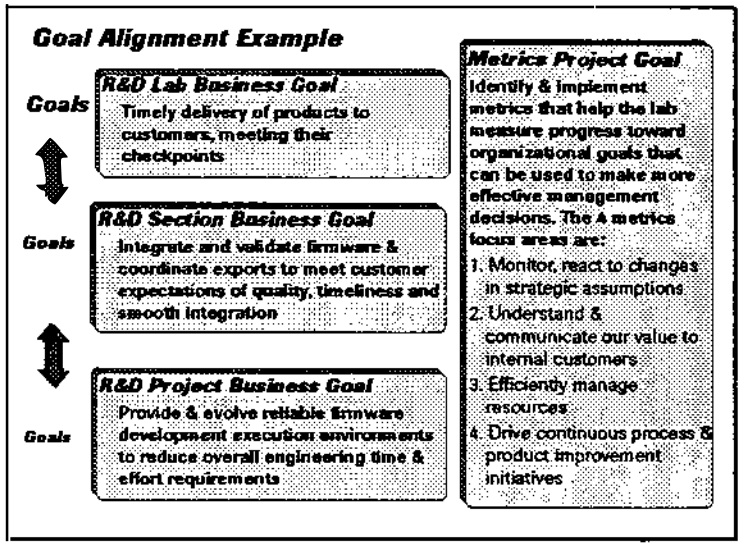



*Figure 11*

The next step was to select a representative metrics team with whom we derived a set of goals for the combined firmware sections. We further assured that they were aligned with the lab-wide goals. In this same half-day session, we brainstormed a total of 150 questions and a sampling of metrics for the six goals. Because one of the goals was clearly a higher immediate priority than all the rest, it was decided to focus our near term attention on that one goal. Figure 11 shows an example of how goals at each level are aligned. The metrics project goals are independent.

In the next months, SWI consultants synthesized the brainstorm data into a working summary of prioritized goals, questions and metrics, ultimately arriving at an initial set of proposed metrics and presentation graphs for the highest priority goal. During this synthesis, much consideration was given to the current metrics maturity of the groups and what a reasonable scoping should be for the first set of metrics.

SWI essentially handed off leadership to the division metrics project champion at this point. After presenting the proposal to mixed reception at a section team meeting and to their partners in product development in the different divisions, the project has come to a virtual halt, due largely to interrupting real schedule pressures. It is scheduled for resumption in August.

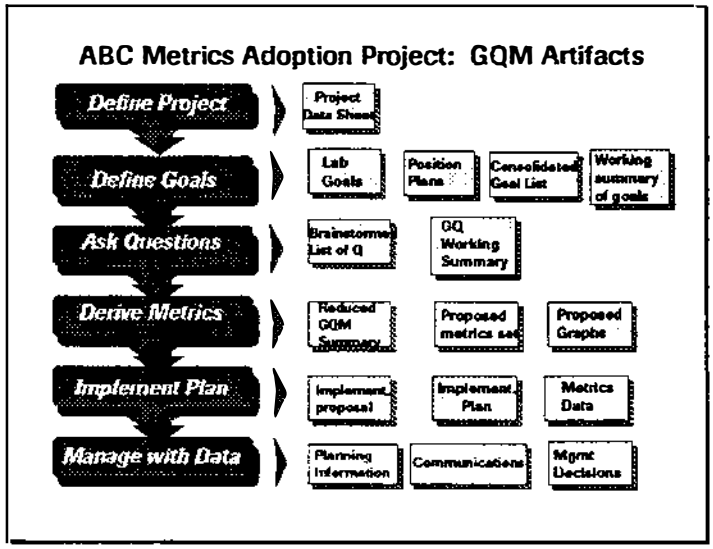



*Figure 12*

The lessons at this checkpoint are many. We have great confidence about the appropriateness of the organizational chunks and the alignment between them. Focusing on the reuse effort already underway was a good motivating factor for the project. Although our assessment of organizational readiness seems fair, we perhaps did not use that information as well as we should have in pacing the project. In particular, the effect of the unconstrained brainstorming process was a feeling of exhaustion on the part of the participants. Despite being assured that the massive list would be synthesized down to a critical few, the process still drained a lot of energy and enthusiasm from the group. On the other hand,

the more thorough process may have had a longer term positive impact on gaining organizational buy-in for the initial set of metrics.

In supporting the implementation process, SWI could also have played a much stronger role in guiding the communication plan. In fact, the initial proposed set of graphs grew rather than shrank after SWI handed it over to the project lead. The presentation audiences were consequently overwhelmed at the size of the effort apparently being requested.

SWI's approach to metrics at XYZ division has been and continues to be significantly impacted by the ABC learnings. XYZ is actually a partner division to ABC and a customer for some of the ABC core firmware. The organizational chunk at XYZ was easily determined to be the firmware section with alignment needed with lab and group level business goals. Although the section was highly enthusiastic about getting metrics in place, they were still at lower levels of the maturity model. A key element of their situation is the adoption of object oriented methods so they want to be able to evaluate the value of the methods. The metrics project objectives are:

- Help to manage the current project
- Improve the planning of future projects
- Measure process improvements and use these measures to drive continuous improvement

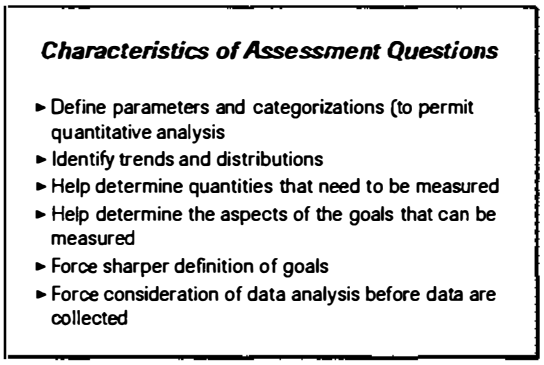



*Figure 13*

The basic GQM steps with XYZ were greatly streamlined. More importantly, we began to prune the GQM tree as we progressed. The seven agreed upon goals were prioritized and the first four determined to be most important. We then developed questions for only those four goals. Additionally, we put some constraints on the brainstorming process in order to arrive at a smaller amount of useful information. Those constraints revolved largely around distinguishing between *assessment* questions (e.g. "Is the time to design a product's software decreasing with use of a reusable framework?") and *understanding* questions (e.g. "How do we test for compatibility?") Giving the team some guidelines for questions that will lead to useful metrics led to a much more effective session and resulted in a shorter, more focused set of questions.

An initial set of 10 metrics was derived fairly quickly. Because the team was comprised of mostly new and inexperienced managers and engineers, however, they perceived serious potential risk in the demands of estimating. Having briefly overpaced our division partners, SWI regrouped, and offered to support implementation of "milestone management," their terminology for monitoring progress against a project milestone plan. The section goals are still in place and SWI will focus its attention on assisting in developing efficient and effective ways of *communicating* about managing the projects.

An additional observation on both projects was the impact of development partner relationships on the GQM process. One of SWI's key contributions to both of these GQM efforts was the mapping of relationships and deliverables between our customer firmware sections and their product development partners. Understanding these relationships was particularly important to the goal setting process.

## Conclusions

Much of what is discussed in this paper is an extrapolation of SWI experiences in other software development areas to software metrics. We hope the validating data will begin to be available in the next few months. We can still draw four general conclusions now.

The first is that all process improvement activity must be outcome-directed if it is to have any lasting effect. In the GQM experience, this means not only directing activities toward well-defined metrics project objectives, but also assuring that the *adoption process* as well as the metrics themselves serve the organization's business goals.

Second, it is the role of the consultant to make that happen, keeping the group focused on the desired outcomes at both project and organization levels.

A third key learning is that the goal is the most important part of GQM. In our experience, establishing and communicating well-formed strategic objectives may be the most significant contribution of the whole GQM process. It has value independent of whether metrics are ever implemented. A more descriptive way of representing the process might therefore be **G**q(m) to reflect the relative importance of each component.

Finally, the pacing of metrics adoption is critical to its success. A grand plan unimplemented has less value than a few simple metrics in action.

---

### References


[1] Grady, R. and D. Caswell, Software Metrics: Establishing a Company-Wide Program, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[2] Grady, R., Practical Software Metrics for Project Management and Process Improvement, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1992.

[3] Ibid, pp. 206-208.

[4] Basili, V., and D. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, Vol. SE-12, no. 7 (July 1986), pp. 733-743.

[5] Basili, V., and H.D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *IEEE Ninth International Conference on Software Engineering*, Monterey, CA (April 1987), pp. 345-357.

[6] Grady, op.cit., p. 208, 211.

[7] Reed, J. "Software Metrics and Policy Deployment", *Proceedings of the Applications of Software Measurement Conference,* Nov., 1994.

# Developing a Low-Defect Software Product in a Research Organization

Jeffrey M. Bell and Laura McKinney*
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
P.O. Box 91000
Portland, OR 97291-1000 USA
http://www.cse.ogi.edu/PacSoft/

**Abstract**

The Pacific Software Research Center (PacSoft) at the Oregon Graduate Institute of Science & Technology has recently concluded a development effort in which the delivered software product had no failures in an extensive usage period by independent users.

We believe the absence of failures is attributable to the appropriate choice of processes, tools, and measurements. Careful selection and adaptation of all three was vital: success with one aspect would not have been enough to ensure the success of the development effort. We describe our key choices: the incremental development process, functional programming language tools. and measurements oriented towards the characterization and prediction of effort and quality.

**Keywords**

Quality, Research Organization, Measurement, Incremental Development, Functional Programming

**Biographical Sketches**

Jeffrey M. Bell (bell@cse.ogi.edu) is Scientific Programmer for the Software Design for Reliability and Reuse project at the Pacific Software Research Center. He holds a MS in Computer Science from OGI and has been developing software professionally since 1983.

Laura McKinney (mckinney@cse.ogi.edu) participated as Project Manager for the Software Design for Reliability and Reuse project at the Pacific Software Research Center and is currently a Scientific Programmer. She holds a MS in Computer Science from OGI and has worked as a software developer since 1981.

*Authors supported in part by Air Force Materiel Command.

# 1 Introduction

The Pacific Software Research Center (PacSoft) at the Oregon Graduate Institute of Science & Technology has recently concluded a development effort in which the delivered software product, comprising approximately 20,000 lines of source code, had no failures in an extensive usage period by independent users (similar to a beta test cycle).

Project goals were both aggressive and challenging. Within a 19 month project timeline, researchers were to explore new research ideas, realize these ideas in a delivered product, and experimentally validate the effectiveness of the new technology using that product. The development group consisted of computer science researchers, all with at least one year of post-graduate education in computer science, and many with Ph.D.'s. While this level of expertise was critical to meeting the research goals, research knowledge alone was not sufficient to ensure a successful software development effort. We believe the absence of failures in the delivered software is attributable to the following three characteristics of the project:

1. Effective processes, especially the incremental development process.
2. Appropriate tool choices, in particular the use of functional programming languages.
3. Measurement oriented towards the characterization and prediction of effort and quality.

Although we describe software development in a research organization, many aspects of the project are common with industrial development projects. The project included external demands that we meet "hard" completion deadlines. The requirements for the system were negotiated with the customer. The customer continually evaluated our progress towards deadlines and achievement of requirements via ongoing informal communication and quarterly progress review meetings.

# 2 The Project

The software was developed in a project known as *Software Design for Reliability and Reuse (SDRR)*[1, 4], conducted for the United States Air Force Materiel Command. At the core of SDRR is a method for the creation of automatic software component generators. The method uses a reusable program transformation "engine" that can be integrated with a domain-specific specification language to produce a special-purpose generator. Program transformations improve the efficiency of the generated code. The reusable engine is configurable for different target environments. All that is needed to create a new generator is a small translator for the domain-specific specification language and configuration parameters to specify the target environment. With this method a domain-specific software component generator can be constructed quickly and inexpensively. Furthermore, SDRR software component generators can be effectively used by problem domain experts who are not software engineers.

In the project we defined the SDRR method, developed the reusable engine, and built one generator as a test of the method. The generator produces code for the Message Translation and Validation (MTV) problem domain [6]. The MTV generator translates specifications of message formats written in a high-level declarative specification language into executable Ada packages suitable for integration with military $C^3I$ systems. The MTV generator, including both the reusable engine and the domain-specific specification language for MTV, was developed in a fifteen month period ending September 1994. The development team consisted of four professors and one post-doctoral researcher each with a Ph.D. in computer science, five graduate students,

a project manager, and a staff programmer. The generator contained approximately 20,000 lines of Standard ML source code and took approximately 4.6 person years to develop, including both research and software development effort.

The final activity of the SDRR project was an experiment comparing the use of the MTV generator with an existing state-of-the-art software development method based on software templates [10]. These templates, developed by the Software Engineering Institute, are skeletal Ada packages containing placeholders that can be "filled in" to produce working MTV code. The experiment was conducted in October–December 1994 by independent subjects who performed development and maintenance tasks specified by the Air Force using both development methods. The experiment served as a hard deadline for the SDRR project development efforts and also focused those efforts—that is, our development goal was to provide the functionality necessary to make the experiment possible. The experiment was successful and supported the hypotheses that the use of generator technology yields gains in reliability, productivity, and flexibility [3].

# 3 Processes

Many processes contributed to the delivery of a low-defect software product, including testing that was focused on usage of the MTV generator in the experiment; reviews and inspections; and defect and change tracking processes. However, we believe the most significant process contributing to quality was the incremental development process by which the MTV generator was built.

## 3.1 Planning the process

At the beginning of the project, the development team participated in planning exercises to design the software architecture for the MTV generator and plan the development and quality assurance processes.

The original generator design included nine separate tools, arranged in a pipeline architecture (see Figure 1). Tools would be executed in sequence, with each tool getting its input from its immediate "upstream" predecessor in the pipeline, and passing its output to its immediate "downstream" successor. The tools included a graphical user interface, three language translators, and five program transformation tools. The pipeline structure minimized the integration effort, and accommodated file-based interfaces between tools which aided system testing.

Because many of the tools required research as well as development, it was possible that some tools might be difficult or even impossible to build. Moreover, it was not clear at the outset how well the tools would operate together. Because of this, we wanted to integrate the tools as early and as often as possible to give us information to guide the development process. In addition, we designed the tools with overlapping critical functionality, anticipating that some tools might not be mature enough for inclusion in the delivered system.

We planned to construct the tools in three cycles. In each cycle developers were to produce and test prototypes of the tools. At the end of each cycle the tools could be integrated to produce a testable generator prototype. Also at the end of each cycle we scheduled activities to critically examine the tools and the integrated pipeline. This facilitated re-planning of the subsequent development cycle.

In the initial planning stage we outlined the specific functionality of each tool prototype, which conformed to a basic structure. We expected to (and did) revise the functionality requirements for each prototype as development commenced. Prototype 1 tools included minimal functionality to support "manual" integration—that is, interfaces between tools might require

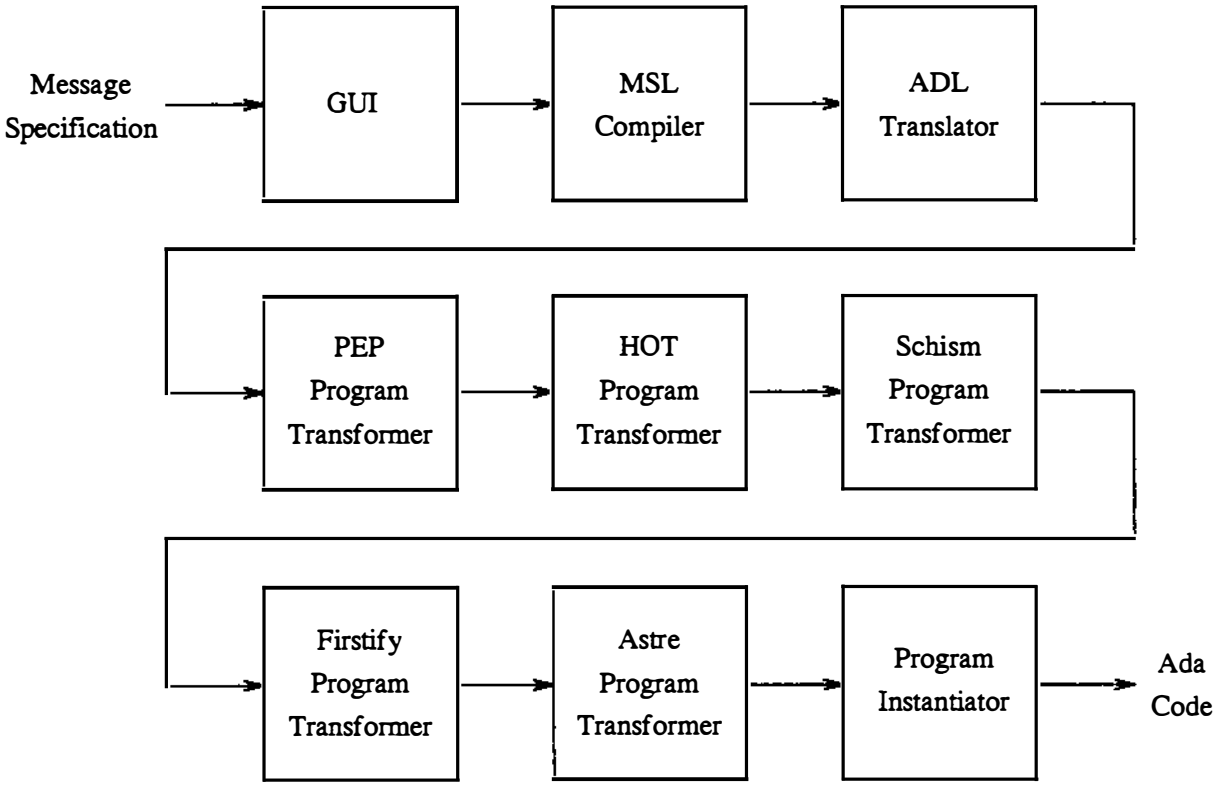



Figure 1: SDRR Pipeline Architecture

some manual intervention to be operable. Prototype 2 tools included minimal functionality to support full integration and the experiment. Prototype 3 tools included additional functionality, such as performance improvements and non-critical features.

## 3.2 Results

Because tools could be integrated at a relatively immature stage, interface mismatches and functionality gaps could be identified early enough to fix them. After the first prototype development cycle was completed, the tools were not fully integrated. Instead, a single example was fed through the pipeline by hand by taking the output of a tool as the input for the next tool in the sequence. At times the output of a tool did not meet the input requirements for the next tool. These interface mismatches were documented and influenced re-planning the next prototype development cycle.

This integration exercise also exposed functionality gaps, since up to this point it had not been possible to run the tools on real data. In fact, it was determined that one tool, the Schism Program Transformer, was not meeting the required functionality for its role in the pipeline. Schism was a tool developed prior to the SDRR project, and our initial assessment of the appropriateness of integrating it with the other tools proved to be incorrect. Furthermore, the developer of Schism had recently left PacSoft, so modifying it to accomodate the needs of the pipeline was infeasible. It was decided that the tool should be replaced with a new tool. Had this early integration not been performed, the difficulties with this tool would not have been seen until it was too late to begin development of a replacement.

The integration of the second tool prototypes produced a fully integrated pipeline. System

testing of this pipeline was more effective in exposing defects, since it was now feasible to automatically run the complete pipeline on realistic examples. Because of the aggressive project schedule, in the re-planning effort following this integration we critically examined what development and rework remained. At this point we decided to not develop the non-critical tool functionality planned for the third prototype. Also, to reduce risk we elected to not include two tools in the delivered pipeline. These tools had already been developed, but experimentation with the pipeline had made it apparent that these tools were not critical. These decisions allowed us to focus on the rework necessary to deliver the system on schedule. Our incremental development process made these decisions possible.

In summary, the use of an incremental development process with frequent re-planning had several advantages

- Tools could be integrated at a relatively immature stage, exposing interface mismatches and functionality gaps early enough to fix them.
- We didn't need everything we planned to develop. As resources became slim, we were able to rationally decide what could be omitted from the final delivery.
- Critical functionality was implemented first.

# 4 Tools

Low-defect software cannot be constructed without the right tools to solve the problem. We found that the best tools for constructing the MTV generator were typed functional programming languages.

Functional programming languages like Standard ML [8, 9] are high-level, declarative, strongly- and statically-typed languages. A key attribute of these languages is the treatment of functions as "first class"–that is, functions can be used as values. For example, functions can be created or manipulated by other functions, passed as parameters, or returned as values of functions. Another key attribute of functional languages is the absence of side effects. As a result of this, a value computed by a function is based solely on the parameters to the function. This simplifies analysis of a function's behavior, and facilitates reasoning about characteristics of a program, since functions can be analyzed in isolation.

## 4.1 The right tool for the task

At the core of a generator developed with the SDRR method are tools that perform meaning-preserving program transformations. Each of these tools is capable of transforming a program into an equivalent but more efficient program. This technology is key to the translation of high-level declarative specifications into executable low-level code.

Functional languages are well suited to the implementation of program transformations. Like many other applications, program transformations can be implemented by defining a data structure (i.e. the abstract syntax of a program) and a set of computations on that data structure (i.e. the program transformations). Many functional languages have recursive datatypes that are ideal for encoding the abstract syntax of a language. Corresponding to each of these datatypes are generic control structures that can be used to perform a variety of computations on values in the datatype. These generic control structures can be automatically generated from the datatype. For example, if the programmer defined a datatype representing binary

trees, a function for performing traversals of binary trees would be automatically generated and available for the programmer to use.

Prior to beginning the SDRR project, PacSoft personnel had experience implementing program transformations with functional languages. Several transformation tools, including Astre and Schism, had been implemented with functional languages. In addition, a functional language based on Standard ML called Compile-time Reflective ML (CRML) [11] was developed by PacSoft in part to support program transformations. This pre-existing infrastructure was further reason to use functional languages.

### 4.2 Advantages of functional languages

There are other advantages to the use of functional languages. Because functional programs are declarative and very high-level, programmers can express computations concisely. Functional programs are also precise at a higher level of abstraction than traditional imperative or object-oriented programs. A programmer writing in a precise, concise language has reduced opportunity to introduce errors, because they are writing less code and the code they are writing is clear.

Functional programs are conducive to formal development methods. In particular, the ability to reason about functional programs improves the effectiveness of formal inspections. The combination of these two factors—the ability to reason and the practice of inspections—encourages the developer to write code carefully and thoughtfully. In the SDRR project we used inspections as the primary means for defect prevention in the implementation of the MTV domain-specific specification language. We are encouraged by the initial results, and plan to use more inspection and formal reasoning methods (e.g. Cleanroom [7] methods) in future development efforts.

We used statically-typed functional languages such as Standard ML as our development platform. Unlike dynamically-typed functional languages such as Lisp or Scheme, statically-typed languages can give the programmer a wealth of useful information at an early stage. This type information assists in both development and debugging. Standard ML programs are written without explicit type information; instead, the compiler performs an analysis of the program to determine type information. This analysis provides the programmer with rich information about which function arguments are actually used, which are used as functions, and which arguments are used in similar or compatible ways. Many functional language programmers find type information a more useful programming tool than a debugger. In fact, all ML code developed in this project was developed without the use of a debugger.

## 5 Measurement

The primary goal of the measurement effort was to provide the data necessary to support rational decision making. Project managers wanted to act early and decisively to control events, minimize risk, and ensure software delivery to support the experiment. The Air Force contract monitors actively encouraged the measurement program, as it enhanced their ability to assess program progress.

Several constraining aspects of the project structure included:

- Fixed delivery date – contractual obligations were made to sub-contractors (who were to act as subjects in the experiment) during a specified time period

- Limited resource pool – specialized skills required of personnel to perform research tasks restricted the availability of "crunch time" resources
- Criticality of late stage effort – the success of the experiment was vital to the success of the project, yet it was necessarily the last effort

These characteristics made it imperative that problems be addressed as early as possible, since alternatives would diminish or disappear as time passed. Tracking project progress alone was not sufficient to ensure a robust product, since problems might not be discovered until late in the development cycle leaving few or no possibilities for recovery. Rather, predictions of future progress were necessary to plan constructively at early points in the project cycle.

## 5.1 Effort

An initial task in the project was the development of an effort data collection system to track effort expended against tasks in the project work breakdown structure. Collection of effort information was done manually, using time sheets filled out by individual developers. To protect the developers' privacy, individuals' reported effort data were protected—that is, management was not privy to any individual's effort information. No performance evaluations, sanctions, or rewards could be based on effort data. This removed any incentive to "adjust" reporting. In addition, the effort reporting system was explicitly defined as a support tool for management to do predictive planning, and not to evaluate personnel.

Since this was a research group within an academic institution, developing a profile of personnel availability using collected effort information helped predict when the software development activities would necessarily yield to academic responsibilities. For faculty, these responsibilities included teaching graduate classes, mentoring students, pursuing other research topics, and participating in department and institutional activities. Students were progressing in academic areas such as taking classes, preparing for the qualifying exams and the research proficiency milestones, and working on Master's and Ph.D. theses. The effort collection system was vital in drawing a group work profile which highlighted areas of maximum and minimum resource availability. Since this was the first project where effort data was collected, the "small" cycles became obvious early and planning could account for those dips. The "larger" cycles which occurred less frequently could only be identified in retrospect. This historical information will continue to be used in further projects to predict available effort.

## 5.2 Capabilities

To measure progress in functionality development, developers reported functionality as a set of "capabilities", which have been likened to unweighted function points [5]. Capabilities are units of functionality from a user's perspective. Identifying capabilities was a rough assessment of functionality and was available early in the project life cycle. No stringent guidelines were set for how capabilities were to be defined, and various developers used different levels of abstraction for identifying a feature as a capability. Developers were asked to identify each capability as either critical or non-critical. Critical capabilities were scheduled for into prototypes 1 and 2. Prototype 3 functionality was strictly non-critical.

## 5.3 Project Scheduling

At project inception, tracking and scheduling was done without the benefit of measurement. The project manager maintained a project schedule developed in consultation with project

personnel. Estimates of completion dates for various capabilities were entered and tracked using a project management tool. The initial versions were entirely based on intuition from experience, as effort data was not yet available. Later versions incorporated known dips in availability which were used to adjust delivery dates.

### 5.4 Prediction of Project Completion

Based on effort and capability completion data, predictions of the final completion date were available as early as 8 months before final release, and remained remarkably stable and accurate throughout the course of the project. These estimates were often slightly more pessimistic than the estimated schedule developed by the project manager and the team, but it was reassuring that the predictions and the schedule were close even though the sources of the data were quite different.

Lacking a historical metrics database, some preliminary development needed to take place to provide enough data upon which to base predictions. We reached this point during the end stages of prototype 1 development, at which time we were able to make our first prediction of the final completion date. Effort expended on completed capabilities gave a rough effort/capability measure. Total effort figures gave a picture of the amount of time available during a given period for development. The two together yielded a rough prediction of the duration of time remaining to complete pending capabilities. Predictions of the final completion date were made periodically and used to validate project schedules as well as identify areas of concern.

In May 1994, with 5 months remaining before the "hard" deadline of the experiment start, a one standard deviation drop in effort expended on project development responsibilities was observed for a several week period (see Figure 2). This decline was primarily due to academic obligations which drew both faculty and students away from project software development for approximately a month. The existence of the measurement data allowed for a rational re-plan at that point, and non-critical capabilities were dropped from inclusion in the final system. As a result of dropping the non-critical capabilities, the predicted completion date shifted back to an acceptable date (see Figure 3). The data allowed for quantification of the functionality reduction, as well as validation that the functionality which remained would be finished in a timely manner. In future projects, this cycle of unavailability which is tied to the academic calendar will be used to make even better predictions.

### 5.5 Evaluation

The effort/capability method of prediction is better as an "early" metric, since differences in individual productivity, capability definition, and project responsibilities have less effect than they might in the last few weeks of the development cycle. The ability to do reasonable prediction reinforces project scheduling and provides a meaningful gauge of effort. Prediction based on capability and effort measurements provides the necessary look into the future while there is time to react.

The existence now of an historical database will allow predictions to be made from project inception, and will provide the basis for assessing personnel availability throughout the year.

## 6 Conclusion

Exploration and implementation of innovative research ideas was crucial to the success of this project, but it was not enough to ensure a successful software development effort. The use

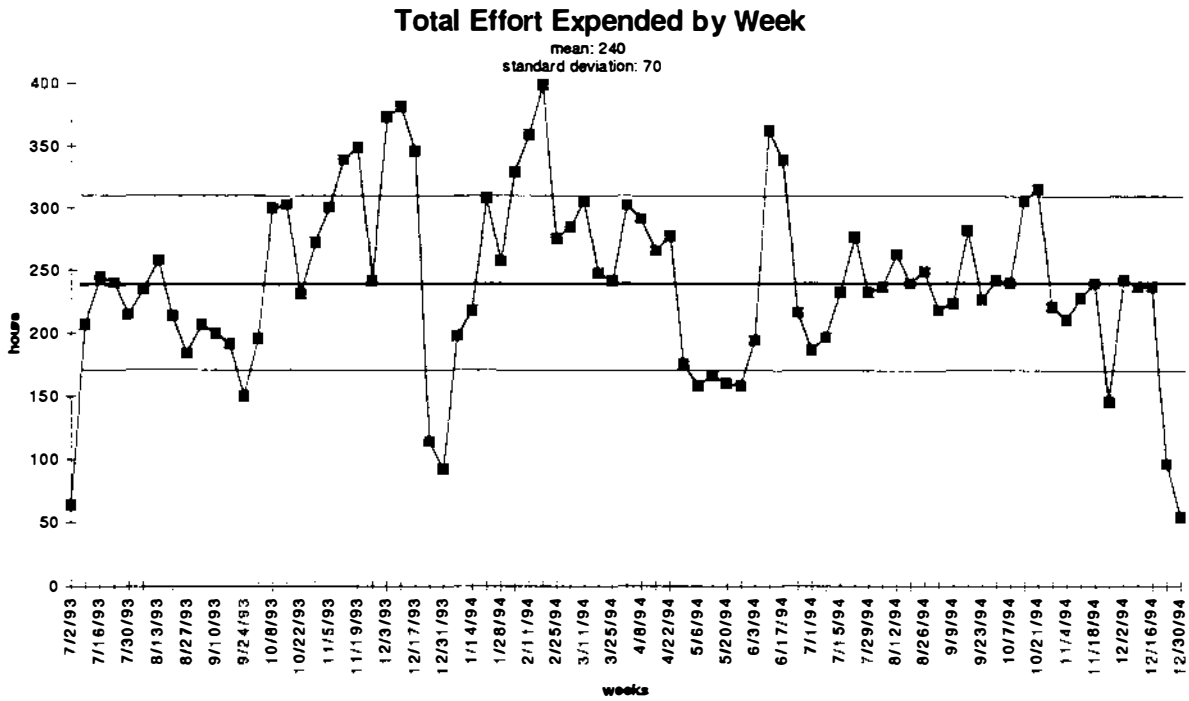



Figure 2: Total effort

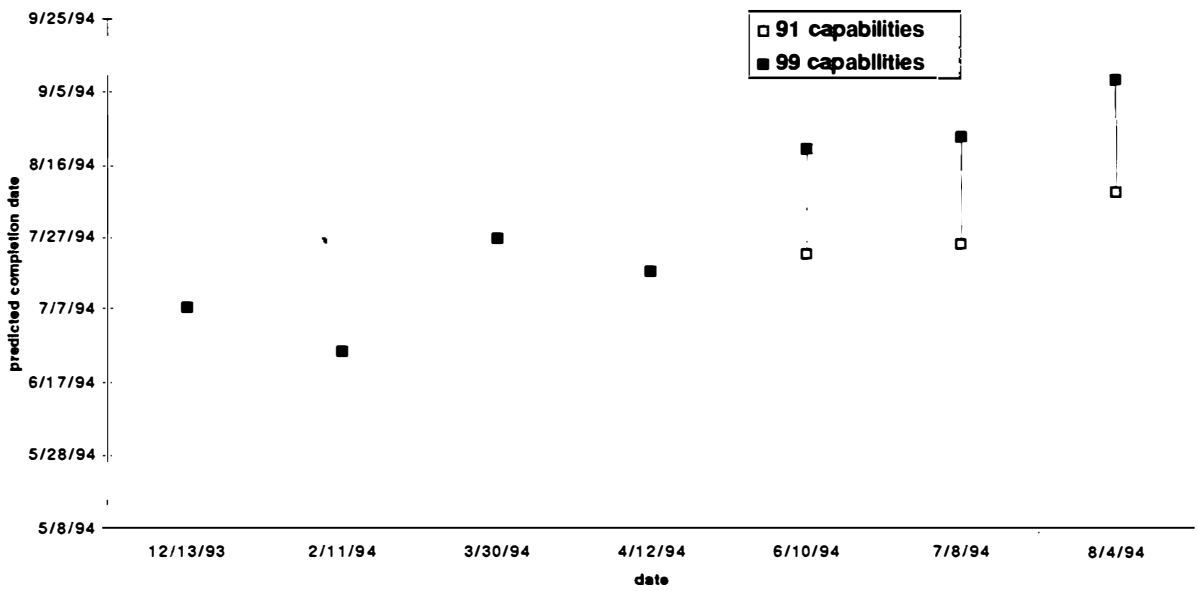



Figure 3: Re-plan impact on predicted completion date

of effective processes to structure the development cycle, the selection of well-suited tools to support development, and measurement to support prediction and management decision making were key aspects in the success of the SDRR project. Independent validation of the software produced by the project demonstrates that this academic organization delivered not only fully-functional software, but robust and high-quality software as well. We believe that attention to the processes, tools, and measurement used in a software development effort can provide great benefits without requiring an extensive commitment of resources. As a small organization with ambitious research goals, the group wanted to make every effort effective. Careful selection of the proper processes, tools, and measurements tailored to the organization was vital: success with one aspect would not have been enough to guarantee success of the project. Good decisions and careful follow-through in all three areas were the necessary combination for delivery of robust software in an academic research organization.

## 7 Acknowledgements

We wish to thank Jim Hook for his insightful comments.

## References

[1] Jeffrey Bell et al. Software design for reliability and reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.

[2] Pacific Software Research Center. SDRR project Phase I final scientific and technical report, February 1995.

[3] Richard B. Kieburtz. Results of the SDRR validation experiment, February 1995. In [2].

[4] Richard B. Kieburtz, Françoise Bellegarde, Jef Bell, James Hook, Jeffrey Lewis, Dino Oliva, Tim Sheard, Lisa Walton, and Tong Zhou. Calculating software generators from solution specifications. Technical Report OGI-CSE-94-032B, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1994.

[5] Alexei Kotov. Application of a new software metric in a research environment. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[6] Jeffrey R. Lewis. A specification for an MTV generator. Technical Report 94-003, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[7] H. D. Mills, M. Dyer, and R. C. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–25, September 1987.

[8] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[9] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[10] Charles Plinta, Kenneth Lee, and Michael Rissman. A model solution for $C^3I$ message translation and validation. Technical Report CMU/SEI-89-TR-12 ESD-89-TR-20, Software Engineering Institute, Carnegie Mellon University, December 1989.

[11] Tim Sheard. Guide to using CRML. 1993.

# A Non-Technical Look at the Evolution of Testing at Visa

prepared by

Anthony C. Alosi
Vice President
Performance and Production Services

Visa International
PO Box 8999 San Francisco, CA 94128-8999
Phone (415) 432-1704
Facsimile (415) 432-3665
Internet Alosi@Visa.com

***Abstract:***
This paper is based on some of the difficult challenges encountered when testing complex Visa clearing and authorization applications between 1983 and 1994. During the period discussed, not only was Visa enjoying tremendous success, but the company was also experiencing rapid organizational expansion and all of the associated growing pains. Many of the issues encountered during testing and the solutions implemented to resolve them extend beyond the walls of the Visa organization. The concepts covered can be applied to any dynamic organization where software development and testing of complex systems occur, where time is always limited and successful implementations are a must. Some noteworthy situations and key concepts are highlighted in this paper.

***About the Author:***
Anthony Alosi joined Visa way back in 1982. Since then he has held several positions of increasing responsibility in the software development, quality assurance and service quality areas. His current duties include management of Visa's Performance and Production Services group which focuses on improving international service quality, Member bank performance and profitability. His previous responsibilities, as Vice President of Service Assurance, included management of software testing for all of Visa's core clearing and authorization applications. Some of his accomplishments include the development of the Visa software testing methodology, development of interactive software testing tools and re-engineering of the software library management, change control and installation processes. Anthony is a member of the Bay Area Quality Assurance Association and is an advisor to the board of the Computer Technologies Program in Berkeley, CA.

## In the Beginning

When I joined Visa in 1982, the Delivery Systems group was a small development organization. In fact, the entire company was small enough that the corporate phone directory still listed everyone by first name. At that time we supported two core business applications which were the heart and soul of the organization: Clearing and Authorizations. These applications were stand alone, never interfacing. Major releases occurred once a year, with sporadic maintenance in between. Although we didn't realize or appreciate it at the time, the development and testing process was relatively simple.

Back then we had no permanently assigned testers or test teams. Developers (we called them programmers back then) did most, if not all, of their own testing. There was no standard testing methodology, each person did his or her own thing. We were small enough that there was no need for a formalized change control process. Anyone could install software into any library at any time. We had a library management system, but it was not widely used. No one really worried too much because the developers were very adept at fixing problems and management was happy to let them come in around the clock to correct production problems.

### Changing Times

As Visa grew, services were continuously added or enhanced. Applications became increasingly complex, inter-dependent and forever intertwined. Implementation schedules began to overlap as multiple versions of many applications were in simultaneous maintenance or development. It was during this period that the need for a more organized approach to software development and testing became painfully clear.

## Method to the Madness

Formalized testing was a relatively new discipline back in 1983 and was virtually non-existent in many companies. We were no exception. For many years we had no formal testing methodology to speak of. The structure of each testing effort was left entirely up to the team assigned the task of testing a given application. Unfortunately, this was usually an entirely different set of players each time around which resulted in significant inconsistencies from project to project and even greater differences from group to group. Many things were left to chance. It was truly sink or swim for those involved.

Over time, we developed a life cycle methodology based on a set of simple guidelines. We realized that within the entrepreneurial Visa culture, volumes of detailed testing procedures, process rules and regulations would not be accepted. By gradually implementing guidelines and not hard and fast rules, we were able to build a framework for success, while still promoting the use of common sense and good judgment among our managers. This gave testing project leaders and managers the flexibility to utilize the framework as appropriate for a given project, but allowed innovative thinking and creativity to flourish. Our testing life cycle methodology mirrored the previously implemented Visa development methodology and was comprised of seven phases (see figure 1).

Testing Life Cycle Methodology

| Phase | Purpose |
|---|---|
| Concept Initiation | Assemble team, gather information, define roles, assumptions list, plan for next phase. |
| Feasibility | Assess risk, identify major obstacles. |
| Test Requirements | Build high level plans, identify environment, data and resource requirements. |
| Test Plan Design | Design test plans and testing environment. |
| Test Development and Execution | Develop detailed test cases/scripts, create test data, build environment, execute tests, write findings. |
| Pre Implementation | Publish findings and conduct formal review, re-assess risk for go/no go decision. |
| Post Implementation | Review the overall project, identify successes/failures identify and implement process improvements, update test environment. |

Figure 1

This meant that for each phase of the development life cycle, we had a corresponding testing phase enabling us to become fully integrated into the project team from the start, rather than at the back end. For example, in the requirements phase, while developers and business people were gathering system and business requirements, we were simultaneously building test requirements. These test requirements would then be built into the overall requirements document, which provided the testing effort much greater visibility very early in the process. We also identified specific deliverables and milestones in each phase which were relevant to testing. We then established a minimum subset of critical deliverables which were deemed required for all projects. Of significant importance were two milestones:

**Test Requirements Sign-off Milestone**
Just as the team would accept ownership of the business and system requirements, this milestone ensured that the entire project team was aware of what would be tested and that everyone assumed ownership. This milestone had a very positive effect on the entire testing team because it gave the testing effort "equal time" in the process. Initially, developers did not take this milestone too seriously, but over time it became an important piece of the puzzle, allowing developers and business office personnel a very good opportunity to review and participate in the test planning process.

**Test Turnover Milestone**
This was a significant change as well. Prior to implementing this step in the life cycle, testing managers had no formal recourse for handling code that was turned over to them prematurely. We were continuously receiving code advertised as ready for system testing, which in reality was barely fit for unit testing. For this milestone, we would establish a minimum set of criteria for acceptance into system test. These criteria were usually established and agreed upon during the requirements phase. This would include successful completion of basic functional tests, production ready JCL, documentation, etc. Once we executed that set of acceptance tests and verified the other criteria, we would jointly review the results and make a decision to either move forward with testing or "return" the code to development. This technique was quite effective during several critical projects. The short term result was postponement of the turnover to QA until developers had a chance to thoroughly unit test the applications. In some

cases this meant project implementation delays. The long term result was much cleaner code and a less frustrated system testing team.

### The Best Laid Plans

Unfortunately, we soon discovered that although this wonderful methodology worked quite well for major projects, it did not work well with ongoing standard maintenance efforts. The entire life cycle for a given maintenance project was anywhere from several hours to a couple weeks in duration and our methodology did not lend itself easily to this short cycle. This was a problem because the bulk of our software installs as well as post install software problems were directly attributable to maintenance. It became critical that we adopt some sort of method to insure maintenance integrity. As a result, we developed an abridged version of the life cycle methodology for maintenance efforts which was also implemented on the development side of the house:

- Test requirements review
- Test development/execution
- Pre implementation review

This abridged version of our methodology gave us flexibility to move quickly, but also gave us enough control to ensure the integrity of what we were installing.

## Great Expectations

Managing expectations was one of the most challenging aspects of software testing at Visa. There seemed to be a general misconception among those who were not intimately involved in the testing process, that everything would be tested every time and that if the code had gone through the testing process there would be no production problems. Unfortunately, this was not reality. Time, resources and code complexity all played a major role in what was tested.

### Act Early

By identifying early in the project life cycle what components would be tested and what *would not* be tested, we were able to modify expectations so they were in line with reality. We found that identifying what *would not* be tested was equally if not more important than identifying what would be covered. This provided a couple of benefits. First, it clearly identified risk and second, it alerted management to the possibility that perhaps the deadline should be re-evaluated.

### Rave Reviews

At first we simply routed copies of the detailed test plans around to members of the team. Initially, we received little response, if any, and we killed a lot of innocent trees in the process. We finally realized that the test plans were just too unwieldy (some weighed fifteen pounds) and overwhelming for already overworked developers and business office staff. So we began to hold reviews with key players focusing just on the high level functions to be tested. This was achieved by developing our test plan matrices so that they clearly defined the function being tested as well as the objective of the test. As a result, we received excellent feedback on our test coverage, which was what we were really trying to obtain. Ultimately, our methodology clearly enabled this process as we began to present our high level test plans during the requirements

milestone meetings. The overall testing strategy could be challenged by the team and any deficiencies identified early on.

**Risky Business**

We also learned that by identifying and quantifying the risk of *not* testing certain aspects of a system, we were much more successful in getting the time we needed to adequately test applications. As any experienced software tester will probably tell you, project managers can be notorious for squeezing test time as development falls behind schedule. Our managers were much more receptive to changes in the schedule when we were able to state the impact from a business perspective if a certain piece of the application failed. We accomplished this by assigning risk factors to all of the components of an application and identifying the specific impact to the business if a certain component failed, including recovery times, potential lost revenues and damage to corporate credibility. If enough critical components were not being addressed due to time or resource constraints and we exceeded a maximum risk threshold, the manager knew that there was enough risk to re-evaluate the situation from a business perspective.

**Reporting**

For the first few years I was involved in testing at Visa, the reporting of test status was also something less than formalized. It was often based more on gut feel than fact, and was rarely, if at all, communicated in written form. With the advent of the PC, numerous project management and reporting tools appeared on the scene. As we learned to utilize these tools, we eventually became more adept at providing meaningful testing status information.

Of particular interest was the reporting surrounding the number of faults we were encountering. This reporting had a very surprising and immediate effect on the development community. Initially, individual developers and managers became wary of us making this information public. This was perceived by some as an attempt at finger pointing or as a way to demonstrate incompetence. Amazingly enough, the overall percentage of faults encountered in testing began to decline shortly thereafter. Although there were a number of other activities occurring simultaneously, we believe this reporting did play a role in improving the quality of the code delivered to the test team by heightening awareness and promoting accountability.

We also learned the importance of identifying, tracking and reporting overall workload efforts rather than just the percentage of tests completed (see figure 2). For example, 80% of the tests may have been completed at a certain point in time. However, let's say those completed tests only accounted for 20% of the entire testing effort. The remaining tests were the most complex, time consuming and perhaps the most critical. If the first 80% had consumed more than 20% of the allotted testing time, then corrective action may be necessary.

**Sample Test Status**

| Function | Total Tests | Passed/ Completed | Failed | Pending | % of Tests completed | % of Total Workload | Total Faults |
|---|---|---|---|---|---|---|---|
| Collection | 100 | 80 | 6 | 14 | 80 | 20 | 15 |
| Settlement | 200 | 10 | 50 | 140 | 5 | 9 | 45 |
| Delivery | 75 | 35 | 10 | 30 | 47 | 85 | 9 |
| | | | | | | | |

**Figure 2**

Another aspect of the testing effort we began to make visible was the effort involved in re-running tests once faults were corrected. Not only did this bring attention to the hidden effort required, but it also revealed patterns for certain developers and components of the applications. This information also enabled us to more effectively plan future testing efforts.

**Finders Keepers**
The most effetive and well received reporting tool we developed was what we called our Findings Document. This was a high level recap of testing events and results prepared in a standard, easy to read format for management. The Findings documents covered all aspects of the testing effort, staying strictly with the facts as they occurred. This document was so useful because it gave the entire management team not only an objective look at the health of the code, but it also highlighted flaws in the entire development/testing process. We published the Findings Document at the completion of each testing effort, usually in advance of the pre-implementation review. This enabled everyone involved to digest the findings in advance of needing to make a go no/go decision.

## Test Team Credibility

When I first started testing at Visa, the function was considered somewhat less than critical and was not universally accepted by all developers. The perception was that testers were less than competent and considerably less skilled than software developers and other technicians. In the early days it seemed that our credibility and value were always being challenged. Any time a production problem was encountered everyone looked to us since we were the last to touch the code. There's an old saying in the music business, "You're only as good as your last record." In the early days at Visa testing, we were considered "Only as good as our last software install". In many instances our last software install wasn't very good which translated to a perception problem.

We soon realized that the credibility of a testing organization was critical to its continued success. Unfortunately, if our testing organization's credibility and perceived value were tied only to squeaky clean software installs, then there was high probability that the group will not be around for long. So we decided that we had to build up our credibility and define our value to the organization in a number of ways.

**Information**
First, we realized that our staff had far more knowledge of the overall system components and operation than any other group. This was due to the fact that we executed the applications in many configurations and situations on a daily basis. Developers, on the other hand, usually focused on one small aspect of a given application and knew very little about its other aspects. Therefore, we began to provide information and training to new developers and other departments within the company who needed to execute the applications we supported. This included information regarding how to execute the applications, what commands were required, shortcuts in setting up environments, assistance in creating complex unit test data, assistance in interpreting Visa Operating Regulations. We began to hold classes to educate new staff on the mechanics and proper procedures for migrating software from development to test and ultimately to the production environment. Not only did this help our audience, but it also

made our lives much easier over time because new people understood the process much sooner than in the past.

### Service

As more and more interrelated applications sprang up, so did the need to perform interface testing with the core applications. Most departments were very busy creating these new applications and did not have the time or expertise to execute the core applications for their interface testing. As a result, many of them tried to forgo this function, but they paid the price at installation time often encountering embarrassing production problems. We saw a need and began to offer interface testing services to various departments. The response was overwhelming and we soon had two full time staff performing this value added function. We began to see a significant drop in the number of production problems associated with interfaces between applications.

### Tools

Not only did interfacing departments need to run interface tests, they also needed a significant amount of test data to perform their unit and integration testing. Once again, there was a severe lack of expertise in creating the complex records required to perform this testing. As a result, we built an interactive test data generator to aid various departments in the creation of test data and execution of the applications.

### Just the Facts Ma'am

Testers are generally inquisitive people by nature. Unfortunately, we all know what happened to the curious cat. On more than one occasion a new and unsuspecting test team member, myself included, would commit what was considered by many of our developers to be the cardinal sin of testing. He or she would step beyond the role of tester and in addition to identifying the problem, would suggest to the developer how to fix it. This practice was consistently not well received and usually immediately created a wall of defensive and adversarial behavior.

Any unsolicited feedback regarding the state of the code would often be misinterpreted as an affront to the developer's technical abilities and problem solving skills. Therefore, we were very careful to instruct our new staff to report only the problem in relation to the functional specifications and not to attempt to debug the code.

### Good Team/Bad Team

Our most successful development and testing efforts always came as a result of excellent team work. We also found that the smaller the team, the better the results. Wherever possible we tried to pair developers and testers together in teams of two or four - almost a "buddy system" to foster the team concept. We built some very good code and strong long term business relationships as a result.

I can recall one example where having this type of team in place provided amazingly wonderful results. At the very beginning of the project we all got together and agreed that our primary goal was to succeed and produce the best quality product possible. Egos would be checked at the door. Even though the task before us was a major re-architecture during which over one million lines of code would eventually be written, we had a common goal from the outset. We assigned a small developer/tester team and empowered them to get the job done. Working within the methodology framework, the team developed a very tight iterative cycle of

development/test/report/repair which gave them optimum turnaround and minimum overhead. They reported their status weekly, and little management intervention was required. In the end, this project encountered the fewest problems of almost any production install I saw in ten years, although the changes were some of the most complex attempted.

In contrast, I still wince when I recall another major project, where the lack of a cohesive team resulted in utter disaster. From the onset, the development management was primarily concerned with control, perception, and boundaries. Several co-project leaders were assigned by the development management and individual developers were specifically instructed not to speak directly with testers. The ensuing a series of miscommunications, cover-ups, delays and frustrations which ultimately led to a total of seven different development project leaders being assigned, huge budget overruns and a delivery delay of over a year.

## I See a Bad Moon Risin'

Each time we tested an application, we learned something. As painful as the lessons sometimes were, they invariably helped us improve the process the next time around. Past history became a close friend as it was an effective tool in sizing, designing and executing testing efforts. It was also helpful in determining if potential problems were on the horizon.

### Place Your Bets

If a particular developer had a poor track record in terms of the number of faults per lines of code, we found that we could expect more of the same the next time around. We learned that the better we knew the strengths and weaknesses of a given developer, the easier it became to anticipate the level of faults in his or her code. Also, if the developer had little or no previous experience with the application, or component of the application, we could anticipate a higher than normal number of faults.

Interestingly, overall programming experience did not appear to be the most significant a factor in predicting the level of faults one could expect from a particular developer. Given equal exposure to an application, we found that new hires, recent college graduates, and entry level programmers often produced tighter, cleaner code than some of the more seasoned veterans. When interviewing some of these "newer" people, it became apparent that more emphasis was being placed on testing techniques in many schools. Most of these junior developers displayed a much more thorough understanding of the testing process and seemed ready to accept ownership of unit and integration testing. Many of the older veterans who did not have the luxury of this type of training often had developed some bad testing practices and sometimes displayed a more myopic view of their role in development environment.

We also began to see patterns emerge during the testing life cycle regarding the number and point in time at which faults were discovered (see figure 3). If the timing or number of faults yielded by the testing effort deviated significantly from this pattern, it either meant that either the developer was godlike, or we weren't getting enough coverage in our testing. The latter was more often the case.

Software Fault Curve

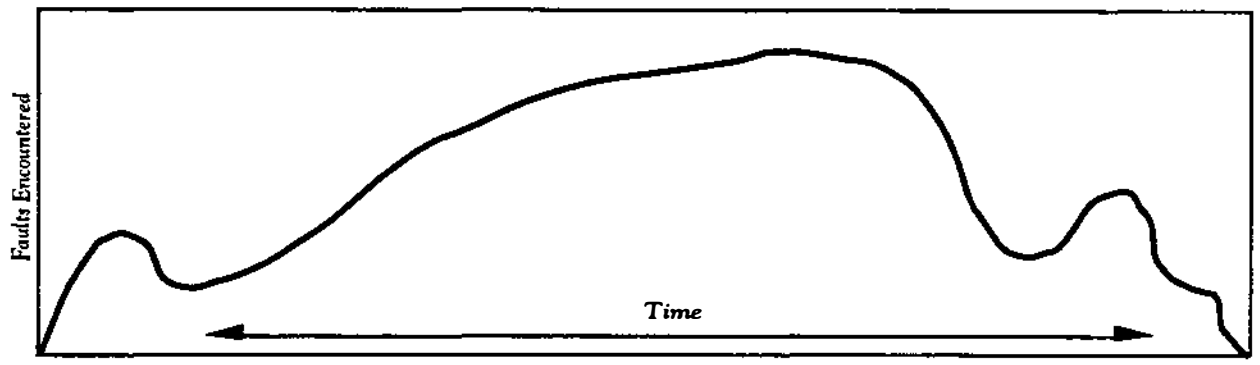



Figure 3

Our own past testing experience with the application itself was also important. As we became more familiar with a particular application, we were able to predict which functions were most likely to fail and therefore we could tailor our testing to focus on those aspects of the application.

## Technology - Friend or Foe?

### Home Grown Tools

One of our first testing tools, an interactive test data generator, was developed in-house starting in 1985. We made several mistakes in developing this tool. One of our biggest mistakes was that we did not sufficiently gather requirements from all potential Visa users. As a result, the initial system did not meet the needs of many constituents, performance was poor, and user friendliness nonexistent. We eventually ended up scrapping the whole approach and starting over with a much more robust set of requirements which met the needs of the user community.

### Purchased Packages

If you are anything like me, you probably receive what seems like a ton of mail from a myriad of software vendors on a daily basis. It is likely that a fair amount of that mail is from test tool producers. With all the glittery promotional material whizzing across one's desk it is easy to be lured into buying a package, especially if your company's purchasing policy is not very restrictive and you've got the budget. A word to the wise - BE CAREFUL. It has been my experience that some companies will try to sell you their product whether it truly meets your needs or not. If it comes close and they can make the sale, some may try.

### To Tool or *Not* to Tool...is That the Question?

If you are searching for a sure way to simultaneously jeopardize your testing effort and damage your career, try interjecting some new automated test tool into the mix at the wrong time. There's an old saying, "Don't Change Horses on Race day." This saying applies not only to the rack track, but can also be a good rule of thumb when it comes to test tools and automation. If you are thinking about purchasing or implementing a package, consider integrating it carefully, and gradually, if possible. Select a project (preferably a low risk project) with which you think you can pilot the tool in a controlled manner. Be sure to factor in additional overhead time for setup, training and false starts.

One of our most successful experiences was with a purchased automated capture/playback tool and came after many months of searching for the right product. I attribute our success to several factors:

**Well Defined Requirements**
We knew what we needed before we started looking. It was easy to weed out the competition and it was harder for them to railroad us into buying something we didn't need.

**Product Evaluations**
We went to trade shows, requested in-house demos, and read all the promotional literature we could get our hands on.

**User Feedback**
We called other users of the products, asking them how they applied them, and what they thought. Users were very helpful and most seemed more than happy to share their experience with us, good or bad. It was particularly helpful when the user operated in a similar environment. We also found that most vendors who really believed in their product and what it could do for us were very willing to provide us with the names of satisfied customers.

**Vendor Viability and Support**
There's another old saying which is often used in the music business, "Here today, gone tomorrow." That saying could easily be modified for software vendors to read, "Here today, gone later today." Be sure the vendor you are dealing with will be around for a while. A little diligence goes a long way. As we conducted our product search, one of our requirements was that the vendor provide extended installation and training support. What we found was that in many cases these items were negotiable. Ultimately, we were able to find a vendor who provided not only the tool we needed, but also excellent support at a minimal cost. We also made sure that any customized support arrangement with the vendor was in writing.

**Evolutionary Approach**
Before we began, we decided to move cautiously. Not wanting to focus only on quick success, we adopted an approach which we believed would also ensure results on a long term basis. After spending a good chunk of the company's change, we knew we would be watched closely and that it would be important that we do things right the first time. We laid out a plan which provided both measurable short term success and a significant return on investment for the company over a two year period. We established incremental and achievable goals over the first six months. We anticipated startup problems and issues and we were not disappointed.

**It Pays to Advertise**
We made sure that everyone we came in contact with knew the success we were experiencing with the product. We were constantly looking for new opportunities to utilize this tool. Within a year, we had far exceeded our original goals and actually had brought several departments on board (including testing and development teams).

## Out of the Ashes

My first major disaster as a test project leader was personally devastating. It was my worst nightmare (the one I'd been waking up from nightly in a cold sweat for about two months) come true. Besides enduring all the finger pointing, chastising, and second guessing, the personal frustration of not having "caught everything" in test came as a big blow to my ego. Fortunately, it was only a "certification " of the production system and no major harm was done. My manager at the time was very experienced and supportive. He helped me to use the experience as an opportunity for growth, rather than self pity and remorse.

**Okay, What Really Happened?**
Post Implementation Reviews were established as a part of our life cycle. They were usually chaired by an unbiased third party and provided a "safe" and open forum for the entire team to express their thoughts about the process. Not only did we highlight the problems, but we also made a point to acknowledge things that worked well. Out of these meetings would come action items which would hopefully help to improve the process the next time around. We would usually hold these meetings well after the implementation had "stabilized" and everyone had the opportunity to reflect on events throughout the life of the project.

As we became more adept at testing our applications, major problems became almost a thing of the past. However, on the rare occasion that something significant did occur, our testers tended to take it to heart. They had become rather accustomed to clean installs and took pride in our success record. Their immediate reaction was often to blame themselves. To preserve damaged tester psyche and motivate them to move forward, we found it was very important that we worked to identify the cause of the problem in a non-accusatory and supportive manner. Once we distinguished their errors from errors in our process, we would then work together to educate the tester and/or improve our process.

## Rewards

Testing can at most times be an utterly grueling and thankless job. If my test teams could have accumulated frequent flier miles for all the weekends and nights we put in over the years, we could have easily booked a few space shuttle trips to the moon and back. Unfortunately, there are no frequent flier miles for testing, and it is up to the testing manager to ensure that the team is consistently acknowledged and rewarded for their efforts. I cannot over emphasize how important this concept was to our success. We took full advantage of the opportunities made available by the company to reward our testers. We did little things that really made a difference. We also made sure that if our people had to put in long hours or weekends, we were there with them. As a result, we had great morale, an incredibly dedicated staff, and one of the lowest employee turnover rates in the entire corporation.

# A PROCESS MODEL FOR MANAGING PRODUCT SUPPORT RISK AND DESIGN QUALITY

Don Moreaux
Hewlett-Packard Company
Boise Printer Division
11311 Chinden Blvd., MS-397
Boise, Idaho 83714

e-mail: dmoreaux@hpdmd48.boi.hp.com
voice: 208/396-5675

**Abstract**

This paper describes the process and results of an endeavor at the Boise Printer Division of Hewlett-Packard to establish a formal connection between our customer support team and the printer product software development team, as part of the software development process. As a result of this endeavor, product quality improved through increases made in our product's supportability, the development process was enhanced by the introduction of an early defect prevention activity, and our support organization improved their ability to prepare for product support by collecting product information far in advance of release.

***Keywords:*** Supportability, focus group, product quality, process management, process improvement

***Biographical Sketch:*** Don Moreaux is a member of the Quality Assurance/Quality Information Technology Group at HP's Boise Printer Division, and a Ph.D. student in Computer Science at the University of Idaho. He began his professional career as a Software Quality Engineer with Varian's Medical Division, before accepting his current position at HP three years ago. While at Varian, he was responsible for testing software control systems for their high-energy linear accelerator, which is used in the radiation therapy of cancer patients. His first assignment at HP was in software quality, working as a testing consultant for the laser printer firmware development teams. In the two years since that time, his responsibilities have shifted to include software metrics analysis, consulting, and training for the printer division. He holds a BS in Zoology and Mathematics from Utah State University, and an MS in Computer Science from the University of Idaho. His research interests include software engineering, software measurement and decision making, and software reliability.

# 1. Introduction

Generally speaking, the practice of software engineering offers two approaches for assuring the quality of any software product. The first approach involves testing some implementation of the software product in an attempt to cause failures in the operation of the software. Failures can then be mapped back to defects in some element of the software product. If correcting the defects eliminates the cause of the failures without injecting new defects, the quality of the product is improved. While both the theory and application of software testing have made great advances over the years, it remains a very expensive activity that occurs late in the life cycle.

Unlike testing, reviews and inspections are *proactive* methods—executed in the early phases of the software product's development life cycle. Reviews and inspections improve product quality by preventing defects from propagating through early abstractions of the product to the product's implementation. While inspections and reviews may never completely remove the need for testing, they offer the additional benefit of uncovering defects at a point in product development when correction is most cost-effective. Less formal review activities can also provide valuable information about certain aspects of the software product's quality. Additionally, the cost of reviews and inspections—if intelligently utilized— are extremely low relative to the cost of adequately testing large software systems.

This paper is written for managers of software projects and software engineers who are interested in using proactive methods to increase their product quality. This paper describes a repeatable and quantifiable process that reduces the uncertainty associated with software design issues through the identification and management of risk, reduces the cost of product support, and enhances the effectiveness of the customer support organization.

## 1.1 Background

Hewlett-Packard's laser printer enterprise is very successful. Since entering the market in the early 1980's, HP has sold over 6 million laser printers world-wide, and currently maintains a substantial portion of the market share. Today's HP LaserJet© product line, having gone through 5 generations of product development, spans a wide range of prices and functionality. Hewlett-Packard has always held a reputation for delivering products of high quality, and our laser printer products are no exception.

The development of the firmware (real-time, embedded systems) and software components of our product occurs in the Research and Development (R&D) section of our Boise Site. Firmware and software qualification and system testing is done within the organization by a separate Software Quality Assurance section. As is often the case in this industry, a substantial amount of our time and resources is expended in the testing process. And, like many other developers, we are continually searching for ways to reduce the cost of testing, without compromising the quality of our product.

It is a widely held belief at HP that the quality of our products results from a planned set of activities, planned early in the product's life cycle. The planning process includes defining customer product requirements, mapping these requirements into identifiable, measurable goals, and providing checkpoints along the development cycle to determine whether these goals have been met. Early goal setting and evaluation allows for early focus on potentially difficult areas, and for early allocation of resources to address these areas. Like many other HP divisions, we use the FURPS model to define these goals, and to help us plan for the quality of our products.[Gra92]

## 1.2 FURPS and Product Quality

HP divisions use the FURPS model to identify the quality attributes used in planning for successful product development. FURPS is a quality acronym that stands for:

- **F**unctionality - feature set, capabilities
- **U**sability - ease of use, ease of learning
- **R**eliability - mean time to failure, severity of failure
- **P**erformance - response time, system efficiency
- **S**upportability - cost of ownership, serviceability, installability

Individual project teams are encouraged to develop objectives for each of the FURPS attributes as early in development as possible, and to do so from their customer's perspective. Though competition, company business goals, and safety and regulatory factors also play an important role in deciding the level of product quality, we recognize that the ultimate measure of quality is our product's value to our customer. Put more simply, value is what remains when our customers subtract the costs of product ownership from the benefits of product ownership. Supportability, a cost of ownership quality attribute, is an attribute we focus on in this process.

## 1.3 Supportability

Supportability incorporates many product characteristics that we closely associate with the product's cost of ownership.(figure 1-1) Supportability considers those product characteristics that make our released product easy to support, and easy to own. Relatively speaking, products that are more supportable exhibit lower maintenance and testing costs, higher degrees of compatibility and installability, and fewer customer calls.

Developing a meaningful and comprehensive set of Supportability objectives requires input from sources outside of the design team. These outside sources must be able to assess our product's *fitness-for-use*—the quality of our product as perceived by our external customer.[Deu88] Since we have millions of printer customers spread over a wide geographical range, bringing them in to help us establish our quality goals is unreasonable. Survey and sample methods are useful user information collecting techniques, but are limited to collecting information about released products. The best alternative is to locate surrogate customers, to act as the voice of the

customer. At our division, we make effective use of surrogate customers in other quality improvement activities such as Usability Inspections. [Mar94]

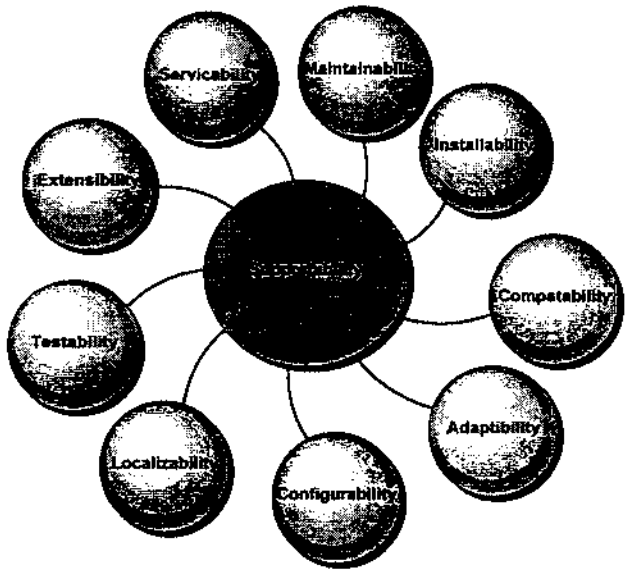



Figure 1-1: Supportability Characteristics

## 1.4 Customer Support

Market share and customer support expense are often used as external indicators of the quality of a company's technical output. [Ste89] While market share provides an indication of the effectiveness of a company's technical effort, it can also be influenced by factors such as inefficient distribution or poor pricing. Support expense is measured in the amount of money and resources required to investigate and satisfy customer complaints. A survey of 282 software producing, US-based companies of various sizes reports support costs ranging from 3.25% to 11% (average of approximately 6%) of net sales.[Wil91] While organizations may view this is a small cost, reducing the value by 1% or 2% can result in substantial savings to companies with high sales volumes.

Customer support information is a valuable diagnostic. In the aggregate, we can use levels and trends to compare ourselves with key competitors. At the issue level, the individual nature of the complaints can provide us with useful information about the various components of our software system. However, because of the complexity of our software systems, characterizing this information to the levels of detail that our design teams require is both expensive and difficult. Our printer firmware systems measure in the hundreds of thousands of lines of code, and this excludes drivers and related software systems.

Agents from our customer support organization spend six hours a day, five days a week on the phones speaking directly with our external customers. They are asked to provide technical

assistance across a wide range of products and product features, and work through customer questions as quickly as possible. To be successful, they must amass a tremendous volume of technical knowledge about each product they support. Additionally, they must be able to access this knowledge and use it to communicate with each customer, at a level commensurate with the user's ability to understand. In time, they become the experts in how our customers view the quality of our product. Acting as a customer surrogate, our support agents are an invaluable source of information for our product design teams.

## 2. The Basics of Getting Started

Methods for improving software quality and productivity typically fall under one of two general approaches: [Art93]

- The Silver Bullet - Promotes major innovations that solve it all at once. This approach advances the notion of the big win—very appealing to American sensibilities. Emphasis is often upon the technological solution.

- The Continuous Incremental Improvement - Encourages slow, steady, and deliberate improvements that incorporate innovations as they arise—the notion of the small win used effectively in Japan. Emphasis is upon the process, as well as the technological solution.

The process we present in this paper is derived from the second approach. It is a small win, it deals specifically with process improvement, and it isn't even close to a silver bullet. In the following sections we provide background information to help explain the process approach we took. It begins with an explanation of why we undertook the effort, and ends with a brief description of how it ties into our software development life cycle.

### 2.1 Why? Reducing Uncertainty to Risk

There is a very significant difference between uncertainty and risk. When you make decisions in situations surrounded by uncertainty, you lack the knowledge necessary to determine the possible outcomes of your decisions and the likelihood of their occurrence. Risk, on the other hand, involves decision making only after gathering such knowledge. Because this knowledge - although certainly not perfect - can be quantified, the associated risk can be managed.

Risk, from a technological perspective, occurs whenever a company advances itself into new markets, products, or technologies, as is illustrated in Figure 2-1.[Ste89] The degree of risk not only depends upon the distance from some existing state, but also upon the number of axes traveled. So, any company advancing a new technology into a new market segment, accepts more risk than if it had simply moved into a new market with its existing technology. Competitive pressure requires us to move from this existing state. To assess and manage the risk, we need information from many sources around our organization. In this paper, we use data from our

customer support center and the expert opinion of our support agents as sources of information to reduce uncertainty and manage risk.

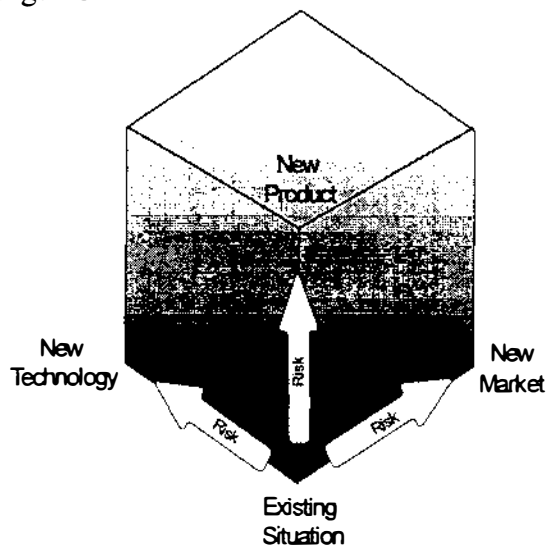



Figure 2-1: The Technical Perspective of Risk

## 2.2 What? Inspections, Reviews or Focus Groups

In addressing how to access our support group's knowledge, it was necessary for us to choose between three review methods. Inspections are a rigorous *in-process*—while the product is being created—review method intended to detect errors in code or documentation. [Deu88]. Reviews, such as formal design reviews and internal reviews, lack the rigor and formality of inspections. While reviews provide project management with a view into the state of the product design, they are not effective for assessing the quality of the software product. Focus groups are also informal, but by their nature, provide an effective format for evaluating product quality.

A focus group is a facilitated meeting that is intended to discern likes, dislikes, trends, and opinions about current and future products. [Bos91] The information derived from focus groups is important because the opinions and desires are captured in the customers' words. The focus group format offered us an opportunity to:

- Avoid duplicating existing review and inspection activities
- Get away from the defect, or fault finding mentality
- More effectively mix support and engineering staff
- Capture collective opinion

While a focus group session may be less formal, it will still require some pre-meeting planning, in-process management, and facilitation.

## 2.3 Who? Roles and Responsibilities

A focus group must involve enough people to adequately cover the key roles and responsibilities of the process. However, like inspections, involving too many people reduces the effectiveness of the activity and makes the process harder to manage. The three business components involved in our process are:

- **R&D** - Our printer firmware and software research and development group. Two members, the project manager and a senior design engineer, represent the design experts in the process. Their responsibilities include identifying the components(s) of the product on which to focus, preparing a presentation of each of those components, and communicating actions taken on any identified issues.

- **CSC** - Our customer support organization. Agents from this group represent the voices of the customer and use their collective voice to provide expert opinion at certain times in the process. They are responsible for identifying call-generating issues and providing recommendations for each of the issues.

- **QIT** - A function of our quality assurance group. One member of this group acts as the facilitator of the process. This individual's responsibilities include facilitating focus groups, insuring adherence to schedule, and maintaining two-way communication between CSC and R&D.

## 2.4 When? Tie To The Life Cycle

The complete value of this process is lost if it cannot be completed before implementation of the design features. The process can begin as soon as the project team has some means of presenting the design of the selected components. Presentation materials used to communicate the design need to be high-level or usage-oriented (see Figure 2-2), not of a type typical of most engineering specifications and requirements documents.

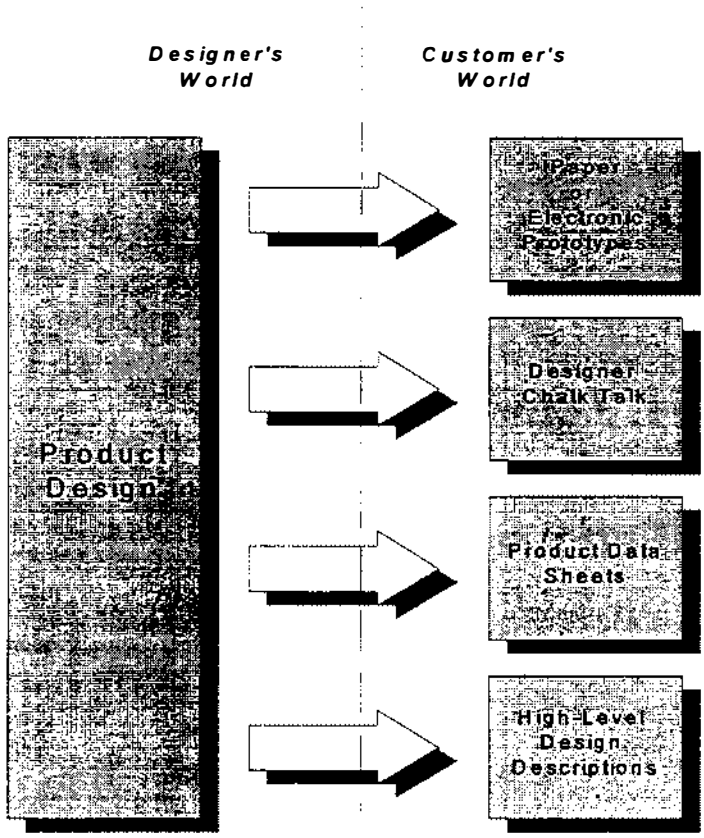



Figure 2-2: Communicating the Design

Our pilot project began the focus group process early in the design phase of our product life cycle (figure 2-3), using *on-line prototypes*—screens, partially implemented utilities, etc.— and design engineer chalk talks to communicate the product design. Since our product was undergoing incremental stages in its design, our activities continued through the implementation phase.

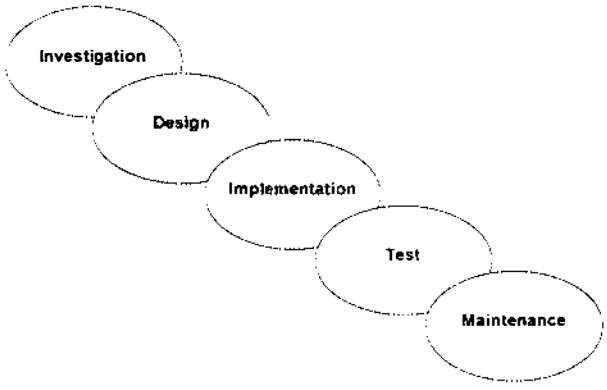



Figure 2-3: Generalized Software Life cycle

### 2.5 How? Process and Change Management

Locating and involving the right people is fundamental toward promoting change in any organization. Understanding the way people view and react to change helped us decide on our process participants. Peoples attitude toward change usually places them in one of three categories: those who promote change, those who are affected by change, and those who resist change.[Sha93] Involving the right people means locating and getting support from individuals who fall into the first two categories. Individuals from these categories can be identified as:

| | |
|---|---|
| *Sponsors* - | are people who have the power and authority to make things happen |
| *Advocates* - | are people who support change, but lack the power |
| *Agents* - | are people with special influence, lack the power, but whose recommendations and suggestions carry weight |
| *Starters* - | are people who initiate change, but by themselves are unlikely to be successful |

Having a sponsor is crucial, and locating one was a high priority before we started any other activity. Of course, piloting the process with a project team composed of advocates and agents also worked to our advantage. To allay the fears of the skeptics, we kept careful measures of the time and effort spent, against which we could later compare with the benefits derived from those efforts.

## 3. Detailed Process Description

Initially we started with only one objective, directed toward deriving a simple estimation of the cost to support several new software features planned for one of our new printer projects. The project manager's request for QA assistance in this endeavor was precipitated by a desire for more information regarding the level of support that the new design features would require. As we worked with our support organization to collect this estimation data, our direct contact with various experienced and knowledgeable support agents made us realize how unique and valuable the information was that these agents collected. If that information could allow us to control - and not just estimate - the cost, it was certainly worth the effort of defining an efficient way to get to it. Time being at a premium for both organizations, we wanted an efficient, yet simple method that would minimize effort, and maximize our chances of collecting meaningful information. The process shown in Figure 3-1 represents a simplified diagram of our approach. The specific details for each process activity are explained in this section.

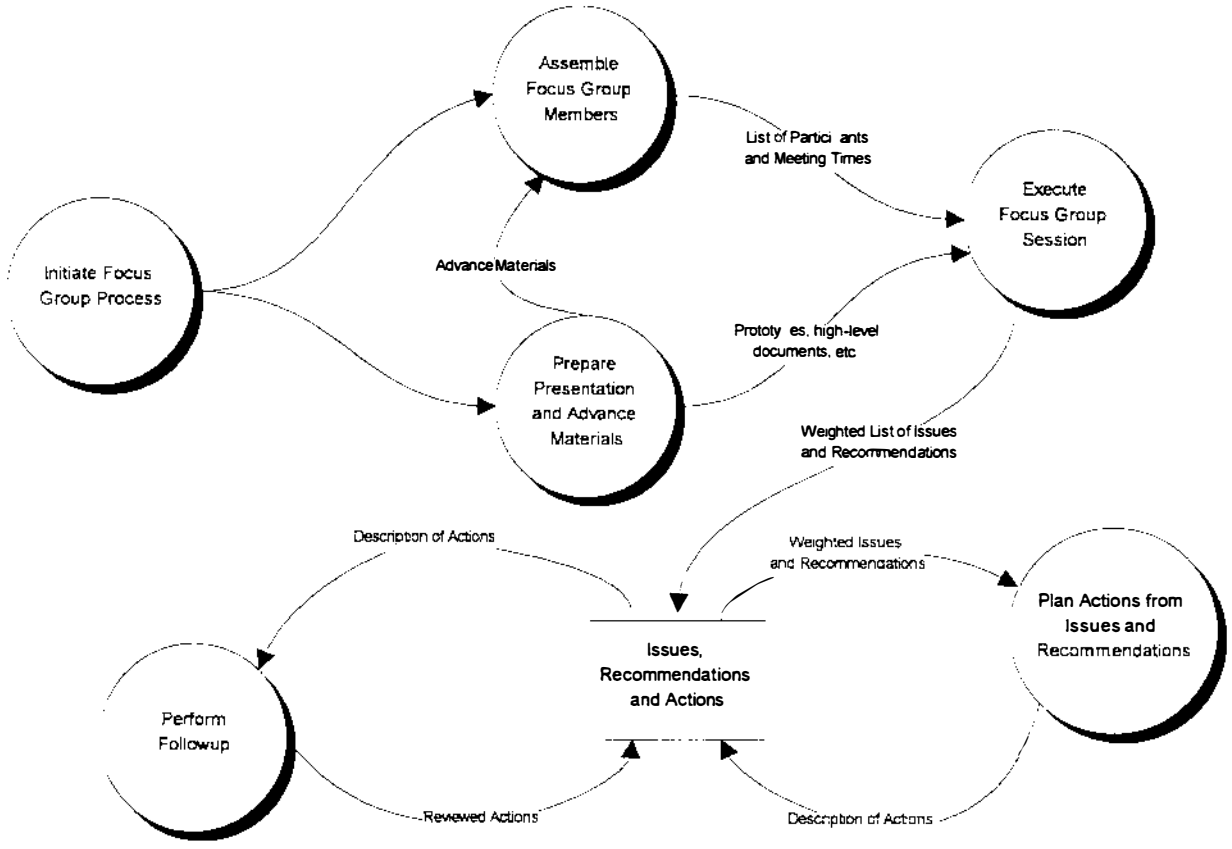



Figure 3-1: Overall Process Model

## 3.1 Initiating the Process

The R & D group has responsibility for initiating the process, with the decision being made by members of the individual design teams. In our pilot, the project manager and the senior designers collectively decided to proceed. In retrospect, they cited the following reasons as the basis for their decision.

- Product (HW, SW, etc.) quality goals include lowering the overall support cost
- The increase in their product's functional complexity could increase support costs
- Quantitative data, not speculation, was needed to substantiate this concern and to drive decisions on design improvements that could offset effects of increased complexity

The process facilitator has the responsibility of ensuring that the reasons are valid, and that adequate time is committed to complete the entire process. If the decision to begin is made too

late in the product's design, engineers will have inadequate time to act on recommendations resulting from the focus group session.

The project team identifies a *base product.* The *base product* is a released product whose basic feature set is similar to the current product. Call-volume data for the base product is collected from our support organization, and later used as the baseline for the final support cost estimation. New product features that will become the focus are identified by the project team. The facilitator takes responsibility for working between the project team and the support team to establish the time line, and identify the milestones and deliverables.

### 3.2 Prepare Presentation and Advance materials

Presentation items include anything that can give support agents a "customer's eye view" or conceptualization of the product from the user's perspective. Items like on-line prototypes, screen designs, story boards, and designer chalk-talks worked well for us. To prepare the support agents in advance of the focus group sessions, we found it beneficial to send advance material about the product. We needed to allow enough lead time before the session for this material to be reviewed, and any questions to be answered. This allows agents time to begin thinking about relationships between this product, and the base product. Advance materials can include high-level design descriptions and product data sheets. Engineering documents may be included, depending upon the technical abilities of the support agents.

### 3.3 Assemble Focus Group Members

Focus group members should include the senior or feature's design engineer, the facilitator, and between 3 to 5 members from the support organization. A membership of 3 to 5 is recommended since research done in the area of formal software reviews and inspections indicates this to be an optimum range for such activities. [Deu88] If possible, enlist agents with base product support experience. When picking agents for the focus group, consider support experience on an equal footing with technical capability.

It is the facilitator's responsibility to communicate the meeting ground rules in advance of the focus group session. At this point in the process it is vital that the facilitator:

- Ensures that everyone can commit the time necessary to complete the process
- Ensures everyone's roles and responsibilities are understood and agreed to
- Communicates clearly the activities involved in the entire process
- Ensures sign-off on the timeline's milestones and deliverables

Since the support agents have the biggest responsibility in commitment of time and deliverables, it's important that they are well prepared. Reviewing the details of the focus group activity with them prior to the focus group session will save time and confusion during the meeting. A brief meeting of 15 minutes between the facilitator and the agents was adequate for our pilot.

## 3.4 Execute Focus Group Session

In our pilot, we limited the design presentations to no more than 1.5 hours, and included on-line prototypes, and chalk-talks given by the design engineer and project manager. Because focus groups deal largely with opinion, we felt it necessary to restrict the types of questions that will be asked. It is the responsibility of the facilitator to discourage questions that invite a design defect finding mentality, or that offer on-the-fly technical solutions to perceived design defects. Our intent is to identify *call-generators*—support issues—and not defects. Questions should pertain to clarification of misunderstandings about the material being presented.

During the design presentation, support agents are responsible for capturing their own set of call-generators. Upon completion of the presentation, the support agents and the facilitator continue on through three sub-activities.(figure 3-2) We suggest that these activities occur as soon after the presentation as possible, while the agents are still definite about the details of the presentation and their issues.

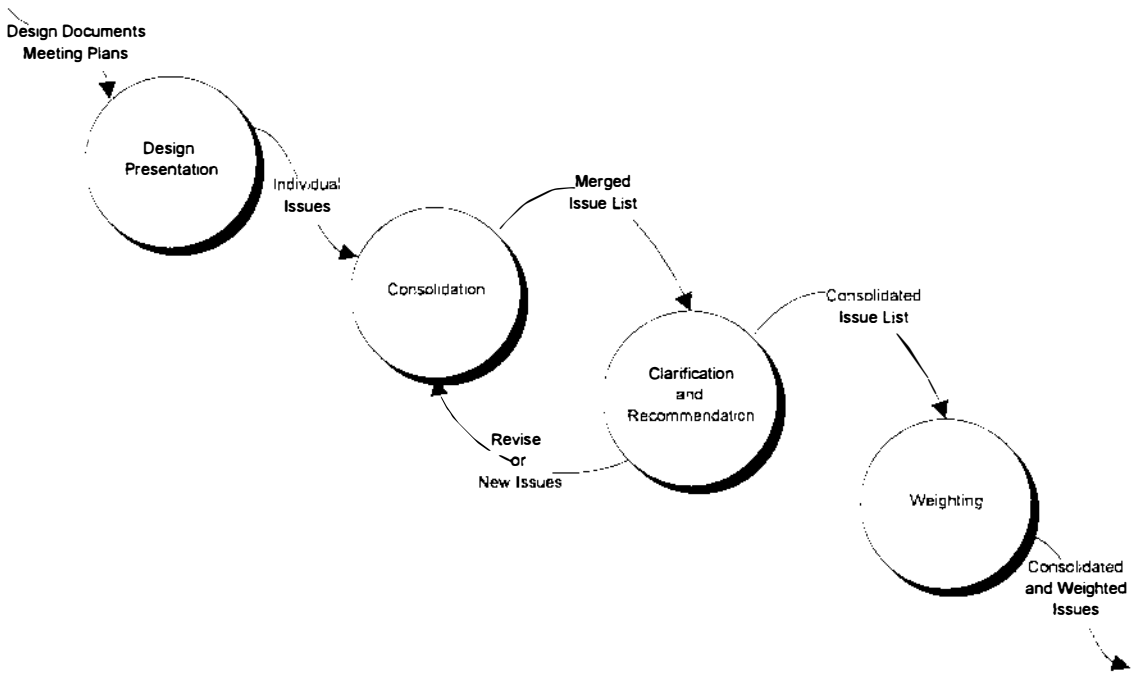



Figure 3-2: Focus Group Session Sub-Activities

The second sub-activity—consolidation—combines the individual call-generators into a single list, representing the consensus of the support group agents. Facilitators have various tools at their disposal to support the consolidation sub-activity. Tools such as Affinity Diagramming or Interrelationship Digraphs are easy to understand and apply directly to group consensus building. [Art93] We noticed a large amount of overlap between each agent's individual issues, making this step a relatively straight forward task.

Having grouped the issues, the facilitator and the support agents attach a clear description and recommendation to each grouped issue. Each description provides feedback to the design engineer, clearly describing the call-generator, and the reasons why the group believes it is an issue. Recommendations are alternatives, derived from the collective opinion of the support team, for avoiding or reducing the impact of each call-generator.

For the final sub-activity, weighting each of the grouped issues, we borrowed a method familiar to practitioners of risk assessment and management techniques. In the classical sense, risk is a measure of the product of two components: exposure and potential. Exposure, as we use it, relates to the breadth of our installed base of users affected by this issue. Potential is the degree of difficulty agents will have working through the effects of this issue with the user.

| | | **Exposure** % User Base | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| **Potential** Length of Call | Excessive | $$$$$ | $$$$ | $$$ |
| | Moderate | $$$$ | $$$ | $$ |
| | Reduced | $$$ | $$ | $ |

Figure 3-3: Risk Assessment Matrix

We found the weighting method to be an effective translating device, bridging the communication gap between the engineering team and the support agents. It communicates the impact of the issue to the engineering teams by providing a means of calculating costs that can be compared with other products, using two metrics that are very familiar to support organizations.

### 3.5 Plan Actions Based on Issues and Recommendations

It is the responsibility of our design team to review the issues and recommendations in a timely manner. Any changes to the product design based on this information is a design team decision. We deliberately limited distribution of this information to the process participants. Since the project team requested the information for their use, we felt they should determine the extent of its distribution. Responsibility for documenting actions taken, or reasons for non-action, lies with the design engineer or manager.

### 3.6 Perform Follow-up

The support staff reviews the issues and actions with the facilitator, and incorporates final comments in the archive. Our support team now incorporates this information into their final estimation of the product's support cost. This final estimation is prepared by adding together the base product data and any costs estimated for the new product.

# 4. Conclusions

## 4.1 Process Benefits

Our time investment for going through the entire focus group process twice amounted to 50 staff hours, with 8 of those hours spent in preparation activities. The first time through, we featured a critical portion of the product user interface. The second pass focused on the product's installation utility. A total of six issues were reported to our design team, with 4 issues generating from the second pass. Of the six, two resulted in changes implemented exactly as recommended, while three others used the recommendation information to implement partial changes. Only one issue received no action as its resolution.

Only one of our six issues received a high risk weighting—High Exposure and Medium Potential—the remaining five were relatively lower. Our estimations indicate the reduction in support cost resulting from design changes recommended by the lowest weighted issue alone will offset our investment of time by a factor of 2 in the first month after release of the product. Based on our estimations, we believe there is enough evidence for us to recommend this process to our other product teams. We plan to closely track the post-release call-length and call-volume metrics for this product, and future products, to validate these claims and ultimately the process.

Perhaps the strongest benefits from the process can be seen in the attitudes of our design engineers and support agents. Both the design engineers and the project manager from our pilot study felt the process was worth the effort—recognized real value in the information they received from the focus group session. They plan to use the process on future projects.

The attitude of our support staff toward the process was even more positive. We were told, after completing the second pass, that the agents who took part in the focus group session volunteered their free time—time when they are not on the phones—to take part in the session.

## 4.2 Lessons Learned

*Never underestimate the value of opinion*

As engineers, most of us would agree that we are more comfortable making decisions based upon quantitative, rather than qualitative information. When qualitative information is structured—resulting from activities such as focus groups—and representative of the customer's opinion of your product, its value should not be underestimated.

*Use facilitation to manage process*

We found facilitation was necessary to insure that deadlines weren't overlooked and that the focus group sessions ran smoothly. All the participants liked the idea of having an owner for the process, one person they could contact with questions and concerns.

Weinberg [Wei93] offers sound advice on running review sessions that as facilitator, I found helpful when facilitating the focus group sessions.

- ù Consensus views are the most valuable, limit your list of issues to these views only.
- ù Don't allow one lone individual to set the tone for the whole group, this will destroy your ability to get consensus.
- ù If the group is reluctant to sign-off on an issue, don't include it with the list. If you cannot get consensus, make sure you capture the information in the archive, but outside the issues list. Differences of opinion are important, they may indicate confusion about the product design. And if the agent is confused, you can bet the customer will be also.

*Consider the Additional benefits*

Technical support organizations have high turn-over rates, losing their agents after an industry reported average of 18 months of service. [Wil91] Unfortunately, 18 months is about the time it takes for agents to fully develop their job skills. The most commonly cited reasons for this turn-over include:

- ù Insufficient training and backup support
- ù Lack of professional rewards, recognition, and career path

We also consider our focus group sessions as training sessions for our support agents. The sessions provide an early opportunity for agents to increase their product knowledge-base, and more adequately prepare for product support. Their professional recognition is enhanced through a process that lets them see that their opinion can influence the design of our products.

## 5. Acknowledgments

Many thanks to Nor Rae Spohn and Doug Howell, our process sponsors, for providing us with the time and resources to work out the details for this pilot. Thanks also go out to Rick Kartes and his support team, with Terry Mahoney and Gary Zimmerman from the design team, who as agents and advocates of change, made this effort a success.

## 6. References


[Art93] Lowell Jay.Arthur, *Improving Software Quality, An Insider Guide to TQM,* John Wiley & Sons, 1993.

[Bos91] James Bossert, *Quality Function Deployment, A Practitioner's Approach,* ASQC Quality Press, 1991.

[Deu88] Michail S. Deutsch, and Ronald Willis, *Software Quality Engineering, A Total Technical amd Management Approach*, Prentice-Hall, Inc., 1988.

[Gra92] Grady, Robert B., *Practical Software Metrics for Project Management and Process Improvment,* Prentice-Hall, Inc. 1992.

[Mar94] Rosemary Marchetti, "Using Usability Inspections To Find Usability Problems Early in the Lifecycle," *Proceedings of Pacific Northwest Software Quality Conference*, Portland, Oregon, October 17, 1994.

[Sha93] Alec Sharp, *Software Quality and Productivity,*. Van Nostrand Reinhold, 1993.

[Ste89] Lowell W. Steele, *Managing Technology, The Strategic View*, McGraw-Hill, Inc., 1989;

[Wei93] Gerald M. Weinberg, *Quality Software Management, Volume 2 First-Order Measurement*, Dorset House Publishing, 1993.

[Wil91] Ralph Wilson, *HELP! The Art of Computer Technical Support,* Peachpit Press, 1991.

# The Quality Improvement Core Team:
## A Grass-Roots Approach to Process Improvement

*Abstract*

*The Quality Improvement Core Team (QICT) is composed of individual contributors including software developers, product application engineers, technical writers, and system verification engineers. The QICT's mission is to act on process and quality issues that affect product development and release at Synopsys. In one and one half years, the QICT has completed one major project and one minor project, and is halfway through another major project. This paper will describe the QICT's projects, what we learned during the projects, and what made the projects successful.*

**Leslie A. Dent**
**Synopsys, Inc.**
**700 East Middlefield Road**
**Mountain View, CA 94043**
**(415) 694-4468**
**email: ldent@synopsys.com**

***About the Author: Leslie A. Dent is a senior software quality engineer at Synopsys. After graduating from UC Berkeley with a degree in Computer Science, she found her niche in studying software quality issues and has spent most of her career in software quality assurance, process improvement and testing.***

# Introduction

The Quality Improvement Core Team (QICT) is an effective method of promoting process and quality improvements at Synopsys. This paper describes:

- why the QICT was formed,
- who the QICT members are,
- who sponsors the QICT,
- how the QICT works,
- which projects we have undertaken,
- what we learned during the projects, and
- what made the projects successful.

The QICT is composed of individual contributors including software developers, product application engineers, technical writers, and system verification engineers. The team's mission is to act on process and quality issues that affect product development and release at Synopsys. In one and one half years, the QICT has completed one major project and one minor project, and is halfway through another major project.

# Background

The QICT was convened by the Director of Product Engineering as a Software Engineering Process Group[1] and then set loose. Since the QICT was formed, it has functioned without direct management involvement or oversight. Synopsys is in transition, growing from a small company with each group having its own formal, informal, or *ad-hoc* software development process, to a larger company that recognizes the need for a standard software development process. At the same time, Synopsys is retaining its small-company culture which values flexibility and grass-roots efforts over mandates. The QICT has positioned itself as a supplier of guidelines, templates, and expertise to the engineering organizations which consist of about 250 people.

# QICT Membership

The QICT was formed with representatives from each functional group that is involved in product development. The functional groups included R&D, Application Engineering, Tech Pubs, Release Engineering and System Verification. The original QICT membership falls into three categories:

- volunteers
- appointees
- recruits

The volunteers have been the most active group members, participating in most project activities. They joined QICT because they wanted to participate in software process and quality improvements.

The recruits have also been active. They were asked to join QICT because a representative from their organizational area was needed and they had demonstrated an interest in software quality and process issues.

Some organizations simply appointed a representative. Most of the appointees have dropped out.

The membership of the group has changed to reflect organizational changes. Synopsys now has four Business Units instead of a central R&D organization. New members have been recruited to ensure that all Business Units are adequately represented. Table 1 shows the current membership of the QICT.

Volunteers are always welcome to join the QICT. We have managed to expand our membership beyond our original sphere of influence.The QICT now includes a representative from the corporate quality program and one from the Customer Support Center.

**Table 1: Current QICT Membership**

| Business Unit/Group | No. of R&D members | No. of CAE members | No. of other members |
|---|---|---|---|
| Design Environment | 2 | 1 | Operations Manager(1) |
| Design Implementation | 2 | 1 | |
| Design Verification | 1 | | QE(1) |
| Product Engineering | | | Product Assurance (1), Release Engineering(1), Software Engineering(2) |
| Support Center | | | (1) |
| Corporate Quality Program | | | (1) |

# QICT Sponsorship

The QICT is sponsored by the Software Engineering group within the Product Engineering organization. Two QICT members are part of the Software Engineering group; one is the QICT leader. Because participating in the QICT is a primary responsibility for these two members, they are able to devote a significant amount of their time to QICT projects. In addition to technical contributions, their activities include:

- conducting meetings,
- acting as project managers,
- compiling the work of the group, and
- giving presentations on the group's behalf.

# Team Mechanics

As an independent grass-roots organization, the QICT chooses its own projects. When the group was formed, it made a list of all possible topics to work on and then voted on a project. Each time the group has voted, a clear winner emerged.

A one-hour meeting is held once a week (if necessary) to plan, review project status, and review project deliverables. Projects are divided into smaller tasks which individuals volunteer for and complete outside the meetings. Sub-group meetings are scheduled and held if necessary.

The official commitment for a QICT member is two hours per week, one hour for the regular group meeting and one hour to work on projects. The level of commitment varies depending on the member's direct management support for being on the team and the member's workload at a particular time. For us, a core group of six does a majority of the work. The remaining members review the work and assist as their schedules permit. The QICT leader prioritizes what they need to review to ensure they give feedback on work related to their area of expertise.

## Product Specification Process Project

The QICT's first project was to develop a standard set of specification templates. Various specification templates have been used throughout Synopsys for awhile, but they were incomplete and not used consistently. Of all of the projects under consideration, the team felt that a standard set of templates would provide the most significant benefit to Synopsys.

### Step One

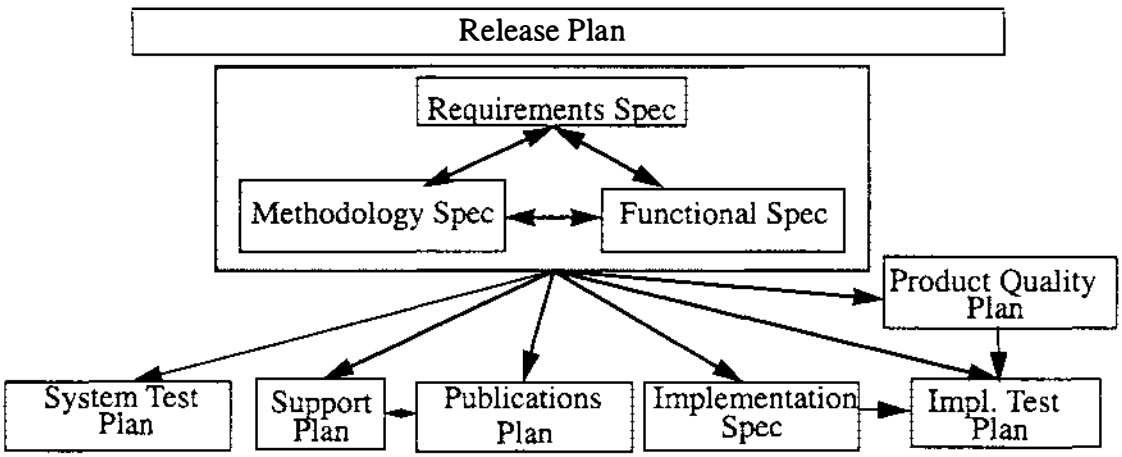
We listed the major problems which we wanted to solve with a new set of specification templates.

- Customer Methodology not considered up-front
- Testing not adequately discussed.
- Porting considerations not documented.
- Support groups unable to plan with information from specifications.

### Step Two

We mapped out which specification templates were needed and in what order they needed to be written. We determined that the information provided by different groups should be in different specifications and that each specification should have a single owner. We also determined that methodology and testing should have separate documents to increase the likelihood of those areas being adequately addressed. Figure 1 shows a hierarchical view of the specification templates.

**Figure 1: Product Specification Template Hierarchy**



**Step Three**

We divided up the templates among the team members and started writing them. Once we started reviewing the templates, we realized that we needed a standard format so that all of the templates would look similar and certain information could always be found under standard section headings. So, we went back and developed a standard format and retrofitted the existing templates to it.

**Step Four**

After the templates were completed, a presentation was formed around them. The presentation included:

- background QICT information,
- motivation for developing the new process and templates,
- benefits of the new process and templates,
- description of the process,
- explanation of each template, and
- emphasis on flexibility (the importance of the templates lies in the information they contain, not the format or division of specs.)

While drafting the presentation, we articulated the process that the templates were part of. We called it a Product Specification process to avoid the impression that we were drafting a software development process. As a grass-roots team, we didn't want to take the responsibility of developing a complete software development process, especially as a first project.

While drafting the presentation, we also took the opportunity to adapt the specification templates and the process to fit different-sized development efforts. When we originally mapped out the specification templates, we had in mind a large development effort like one undertaken for a new product. Many development projects at Synopsys don't fit that mold. Often, a new feature is developed for an existing product. So, we combined templates and came up with a set of two specification templates for a medium-sized project and a single specification template for a small-sized project.

The two templates for the medium-sized project are a Functional Specification and an Implementation Specification. The Functional Specification contains all of the information from the Functional Specification as well as all of the appropriate information from the:

- Requirements Specification
- Methodology Specification
- Support Plan
- Publications Plan, and
- Product Quality Plan.

The medium-sized Implementation Specification template contains all of the information from the Implementation Specification and the Implementation Test Plan.

The single specification template for a small-sized project is called a Project Specification. It is based on the Functional Specification and contains the appropriate information from all of the other specification templates.

### Step Five

The presentation was taken on a "road show" to each product development extended core team. (An extended core team at Synopsys consists of a product marketing manager, a product application engineer, a product technical writer, a system verification engineer and all of the software development engineers who work on the product.) The presentation was given by the QICT leader with help from the QICT representative for that particular group.The presentation lasted about an hour, with ample time for discussion. By having all of the extended core team in one room, the QICT was able to make the audience recognize their dependencies on each other for information and demonstrate how the new process could streamline the communication of that information.

The presentations met with mixed reviews. Some people came in skeptical and left skeptical. Others left with knowledge about new tools to help them in their daily work. The most frequent comments were:

- The requirements change too often to be documented.
- A review or some enforcement to ensure that the right information with the correct level of detail is put into the specifications is needed.
- The Implementation Specification should not be written until coding is finished.
- The templates should be used on a pilot project before being rolled out.

The templates are now being used by a majority of the software developers (16 out of 22 product development core teams) and are updated quarterly from users' feedback. One group reported that they would have still been fighting functionality issues at code freeze if they had not written a Functional Specification up-front. Several groups are not using the templates, but are designing their own templates using ours as a baseline.

As a follow-up project, we are planning to provide examples of specifications which were written using the templates.

The QICT has not gone unnoticed by the executive staff at Synopsys. For its efforts on the Product Specification Process project the QICT received a cash bonus award as part of the Return on Quality Incentive program.

### Lessons Learned

The Product Specification Process project took eight months to complete. We didn't set deadlines for completing templates until several months into the project. We also made the mistake of jumping into the project without specifying all of our deliverables. We understood the goals of the templates but we put the process around them afterwards. We also didn't discuss a roll-out plan until we were done with the templates. If we had managed the project better, we would have agreed upon the schedule and deliverables up-front and planned our roll-out sooner.

### Success Factors

- The emphasis on flexibility, encouraging users to adapt the templates for their needs have made the templates more acceptable to users.
- Many groups have recognized the need for standard specifications. The QICT had the right product available at the right time.

## Specification Review Guidelines Project

After completing a large first project, the QICT decided to tackle something smaller for the next project. To complete the product specification process, the QICT drafted specification review guidelines for holding formal specification review meetings. The project deliverables were:

- a brief set of guidelines, and
- review checklists for each specification.

### Step One

The guidelines, based on the teachings of Freedman and Weinberg,[2] spell out the roles of the moderator, recorder, and reviewers.They also give hints about how to select reviewers. The guidelines state the purpose of the review: *"to identify real problems and potential problems with a specification."* The guidelines also state that the review should only identify problems and is not a place to discuss solutions to the problems.

### Step Two

A review checklist was written for each specification template. Each checklist is divided into three sections:

- content
- completeness
- clarity.

Some questions are generic and apply to all templates. Other questions are specific to the specification template. The checklists were incorporated into the specification templates.

### Step Three

A separate roll-out was not planned for the review guidelines. Several slides about the review guidelines were, however, folded into the Product Specification Process presentation. QICT members have agreed to act as moderators for groups that want to try the review technique.

Some managers especially liked the guidelines because they were succinct and only two pages long. The groups who have used the review technique have found that it is a more effective and efficient way to get the feedback they need. Unfortunately, not many groups have attempted to conduct reviews using the guidelines. Many groups still use the informal review method of passing out specifications and setting a deadline for comments. Those groups haven't completely ignored the guidelines, however: they have used the checklists.

### Lessons Learned

We managed the project better by determining our schedule and deliverables up-front. The Specification Review Guidelines project was completed in two and a half months. However, we decided that we could just use a low-key roll-out plan and let people know the guidelines were available.

[I believe the guidelines are not being widely used because we did not effectively sell them. We should have done a "road show' instead of a low key roll-out.]

**Success Factors**

- The guidelines are simple.
- The guidelines are short.(two pages)
- The guidelines are easy to implement (with QICT members acting as moderators for first-time users).

# Testing Strategy Project

We are currently developing a well-defined testing strategy to be used during software development. The deliverables are:

- a glossary of testing terms,
- an overview of the testing strategy,
- a user's guide which describes each type of testing,
- test plan templates, and
- a roll-out plan for the strategy.

**Step One**

While the two QICT representatives from SE have a strong background in software testing, most QICT members don't have any formal training in software testing. To give everyone a quick background in software testing, two testing summaries were handed out.[3,4]

**Step Two**

A draft of the glossary is complete. The purpose of the glossary is to establish a standard vocabulary to use when talking about testing. Any new terms that we identify later in the project will be added to it.

**Step Three**

The overview of the testing strategy has also been completed. We have chosen to approach testing by dividing it into seven types. See table 2. The seven types were determined in several brainstorming sessions. The types reflect the way the QICT members think about testing their own software. The QICT members are a cross section of the audience for the deliverable, i.e. the people who will be performing the different types of testing.

**Table 2: Types of Testing**

| Type of Testing | Goal | Examples |
|---|---|---|
| **Static** | Reduce the number of errors introduced during requirements analysis, design, and implementation. | spec review, code review, test plan review, user documentation review, code analysis |
| **Functional** | Ensure the code under test reflects the specification. | unit, feature, documentation, error message, bug fix |

**Table 2: Types of Testing**

| Type of Testing | Goal | Examples |
|---|---|---|
| **Integration** | Combine the units of code and ensure they work together. | module integration, intra-product integration, inter-product integration |
| **Performance** | Examine the characteristics of the product under different conditions. | performance, Quality of Results, stress, benchmark |
| **Existence** | Determine if all products in the build can be invoked. | license, existence/confidence |
| **Regression** | Ensure that existing functionality (still) works on all platforms. | functional, integration, customer use, performance, bug fix |
| **Customer Use** | Ensure that customers can efficiently utilize the tools within their design process. | system/methodology, ease of use, beta, installation |

**Step Four**

The user's guide has been started. It will be organized into chapters for each testing type. Each chapter will explain:

- rationale (including risk assessment and cost benefit analysis),
- timeframe recommendations,
- limitations,
- deliverables,
- metrics (to measure testing success),
- tools,
- references,
- testing activities,
- examples, and
- recommendations for future work.

The user's guide will allow groups to decide what types of testing they should be doing, how much testing they should do, and how much the testing will cost.

**Step Five**

Instead of waiting until the end of the project to sell our guidelines, we are trying to get buy-in at major milestones from key managers within the Business Units. We have identified an extended team. With them, we have reviewed the project description and goals and the overview of the testing strategy. In our meetings we have asked for their feedback on our approach and their suggestions for our roll-out plan. Our overview has been well received and we have been given many good suggestions for our roll-out plan. Groups who develop software for internal use also have expressed interest in our testing strategy project, so we are working to include them and their needs.

**Step Six**

A preliminary roll-out plan has been drafted. It consists of:

- doing a presentation of the testing strategy overview to all Synopsys extended core teams
- making all project deliverables available on the world wide web
- developing a testing seminar for each type of testing and presenting it to the appropriate functional group (the group who would be doing that type of testing)

**Project Status**

We are about half-way through the project. It is still too early to determine how successful the testing strategy project will be. However, we have had to fine-tune our management of the project. The project is large and the group is dependent on the time of volunteers so we have had to strike a balance when tracking the schedule. Many sub-tasks have not been completed on time. We have had to replan while trying to keep the group from getting discouraged about missing the original project deadline.We have also tapped additional resources outside the group to work on particular subprojects.

# Conclusions

A Quality Improvement Core Team can effectively promote process and quality improvements in organizations that encourage grass-roots initiatives.

**Keys to success**

- Finding individual contributors who feel strongly about quality and process improvement.
- Recruiting representatives from all software development groups and functions within the company.
- Modifying the membership as necessary to keep current with organizational changes.
- Picking process improvement topics which are relevant and timely.
- Having a team leader whose primary job is the team and who can do what is necessary to further the efforts of the group.
- Effectively selling the QICT's results to software development groups.
- Emphasizing guidelines and flexibility.
- Actively managing our projects.

# References

1. Humphrey, W.S. *Managing the Software Process*: Addison-Wesley Publishing Co., Inc., 1989

2. Freedman, Daniel; Weinberg, Gerald *Handbook of Walkthroughs, Inspections, and Technical Reviews:* Dorset House Publishing, 1990

3. Miller, Edward *Introduction to Software Testing Technology*; Tutorial: Software Testing & Validation Techniques: IEEE Catalog No. EHO-138-8, 1978

4, McConnell, Steve *Code Complete*: Microsoft Press, 1993, Chapter 25.

**David N. Card**
*Software Productivity Solutions*

122 Fourth Avenue
Indialantic, FL 32903

407-984-3370

*David N. Card* helps governments and corporations throughout the world to obtain strategic advantage through the application of software measurement and process improvement concepts.

Mr. Card joined Software Productivity Solutions (SPS) in 1994 in the role of technical leader of the SPS team providing consulting and training services in software measurement to government and commercial clients. Prior to joining SPS, Mr. Card served for six years as the Director of Software Process and Measurement for Computer Sciences Corporation. He spent one year as a Resident Affiliate at the Software Engineering Institute. Mr. Card worked for seven years as a member of the research team of the NASA Software Engineering Laboratory. He has also worked as a software developer, maintainer, and quality assurance officer for large scientific data-processing systems.

Mr. Card received an interdisciplinary Bachelor of Science degree from the American University in 1975, then performed two years of graduate study in applied statistics. Mr. Card has authored more than 30 papers and a book, *Measuring Software Design Quality* (Prentice Hall, 1990), on software engineering topics. He has served as a member of the Editorial Board of IEEE *Software*, Associate Editor of the *Journal of Systems and Software*, and a member of the Editorial Advisory Board for *Information and Software Technology*. Mr. Card was General Chair of the ASQC Conference on the Applications of Software Measurement (1994), Program Co-Chair of theIEEE Conference on Software Maintenance (1993), Program Chair of the IEEE Software Engineering Standards Application Workshop (1991). He also served as Chairman of the AIAA Technical Committee on Software Systems (1992-1993).

Mr. Card is listed in *Who's Who in Science and Technology.* He has received several awards for his professional activities from CSC, NSF, NASA, IEEE, and AIAA.

# Learning From Our Mistakes

Dave Card
**Software Productivity Solutions, Inc.**
122 4th Avenue
Indialantic, Florida
(407) 984-3370

SPS

1

## Learning

- **What to expect - Prediction**
- **When to worry - Control**
- **How to prevent mistakes - Improvement**

SPS

2

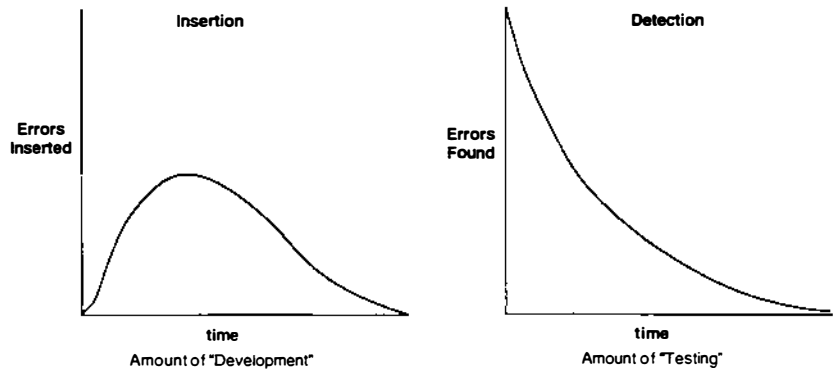
## Error Management Techniques

- **Learning**
  - Prediction - Effort/Insertion Models, Reliability/Yield Models
  - Control - Orthogonal Defect Classification, Control Charts
  - Improvement - Defect Causal Analysis
- **Removal**
  - Testing
  - Inspections

SPS

3

## Two Types of Error Models




SPS

4

## Defect Causal Analysis

- **A technique for continuous quality improvement**
- **A process by which the development team analyzes representative software problem reports to**
  - identify systematic causes of errors
  - propose changes to the process
- **Proposals are then given to an action team for implementation**
- **DCA is not an inspection**

SPS

5

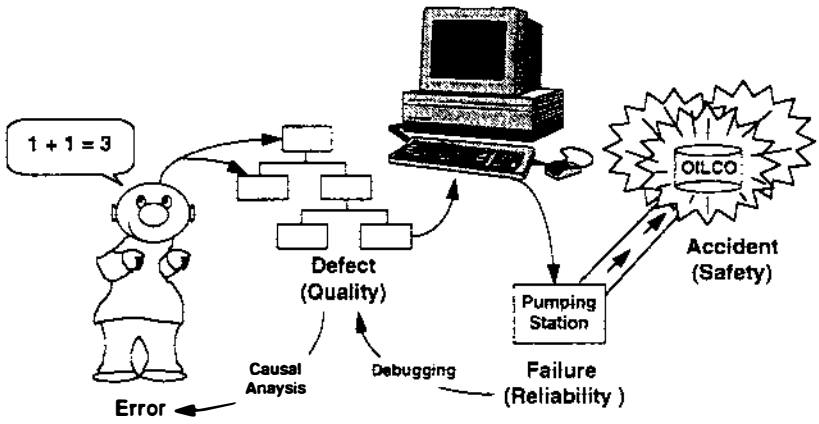
## DCA Approaches

- **Potential information sources**
  - Opinion
  - Basic problem report information
  - Classification data
- **Potential Analysts**
  - Management
  - Quality assurance
  - Software engineering process group
  - Researchers
  - Software engineers

SPS

6

## The Defect Causal Chain




SPS

7

# Causal Analysis Approach

SPS

8

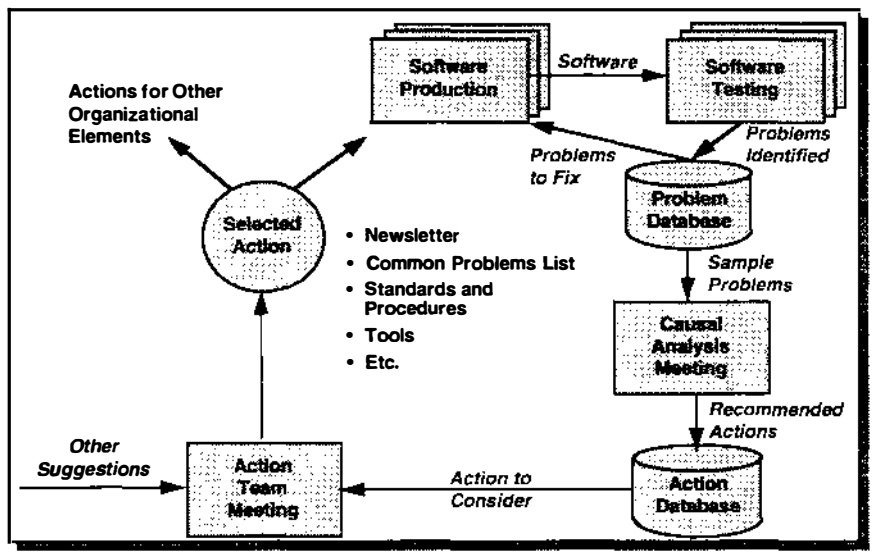
## Concept of Defect Causal Analysis

- Recognize that defects are one of the best indicators of lack of quality
- Assume that software engineers are experts on preventing defects because they best understand the processes that led to the problems
- Keep records of defects (already in place for testing)
- Periodically evaluate accumulated problem reports (usually at end of phase, release, or project)
- Develop strategies to prevent or identify systematic problems earlier

SPS

9

## Defect Causal Analysis Process




SPS

10

## Causal Analysis Meeting

- At least at end of each build
- Approximately 2 hours long
- Involves entire development team
- Designated moderator
- Managers not present
- Open and constructive, not defensive
- Uses structured problems during techniques


11

## Action Team

- Must include management
- May include technical personnel
- Selects, prioritizes, integrates, initiates, and monitors action implementation
- Benefits of DCA are lost without timely action


12

## Example of DCA Process

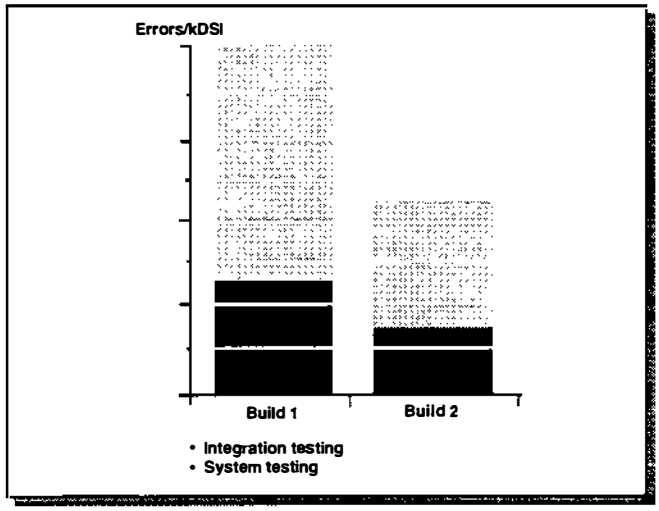
| | |
|---|---|
| ***Problem:*** | Inconsistent use of environment by developers resulted in many errors during integration |
| ***Proposal:*** | Define operational environment (e.g., directory structures, protections) as early possible and perform all testing within copies of this environment |
| ***Results:*** | Integration time for subsequent builds was reduced 50 percent |


13

## *Actual DCA Results*


14

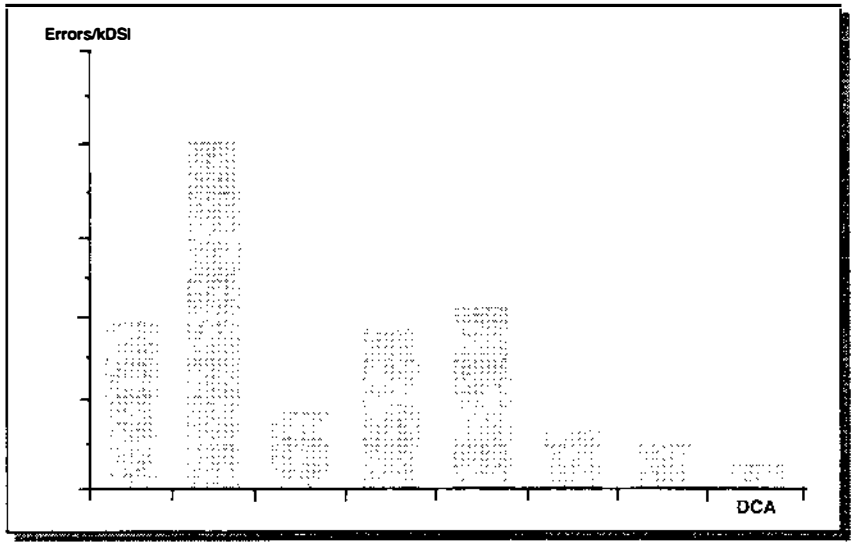
## Comparison of Error Rates for Builds 1 and 2




SPS

15

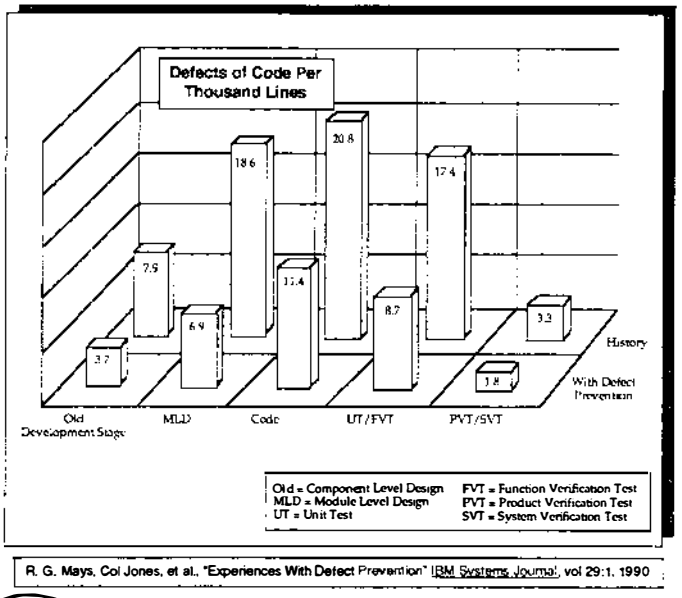
## Development Product Quality Comparisons




SPS

16

## Evidence Of Defect Prevention




R. G. Mays, Col Jones, et al., "Experiences With Defect Prevention" IBM Systems Journal, vol 29:1, 1990

SPS

17

## Maturity-Pull Effect of DCA

- **SEI CMM is descriptive not prescriptive**
- **Level 1 organizations usually implement SEPGs and training (level 3) to get to level 2**
- **DCA helps organizations trying to establish themselves at level 3**
- **DCA does not fully satisfy the defect prevention KPA of level 5**

SPS

18

## Sources of Systematic Errors

- **Methods: 65%**
  - Failure to follow defined process
  - Failure to communicate information
- **People: 15%**
- **Input: 12%**
- **Tools: 8%**

SPS

19

## Summary of DCA Experience

- **Easy to implement**
- **Low cost (≈ 1.5 %)**
- **Increased awareness of quality, Process, and measurement**
- **Tangibly improved product quality**
- **Personnel reacted favorably**
- **Easily adapted to other activities**

SPS

20

# Cleanroom Software Engineering and "Old Code"— Overcoming Process Improvement Barriers

**Michael Deck**

CLEANROOM SOFTWARE ENGINEERING, INC.
BOULDER, COLORADO

## Abstract

One of the greatest barriers to software quality and productivity improvement is "old code." University curricula, CASE tools, and the creative desire of programmers all focus on writing new programs from scratch. The reality of software engineering, though, is that the majority of software work lies in improving and maintaining existing systems. Cleanroom software engineering is based on the principle that **defect prevention is more cost effective than defect removal**. Cleanroom has a track record of quality and productivity improvement in a wide variety of project types and environments. The early Cleanroom papers and reports contributed to an "only for new systems" myth. In actual application, however, Cleanroom provides a number of practices and techniques for improving quality when modifying an existing system. Drawing on published results and the author's own experience, this paper describes how Cleanroom practices are used for existing systems, including small modifications, partial rewrites, and adding new components. It also suggests how, through these techniques, Cleanroom can be introduced even after the development process is underway.

## Author Bio

Michael Deck is a Cleanroom software engineering consultant. He began his professional career as a programmer at IBM on the first-ever industrial application of Cleanroom software engineering. In the 11 years that followed, he was involved in both the application and evolution of the Cleanroom approach, serving as developer, team leader, manager, and methodology consultant. In 1993 he formed Cleanroom Software Engineering, Inc. to assist companies with the phased, tailored introduction of Cleanroom practices. His clients include small businesses as well as multi-national corporations. He is the author of several papers on the theory and practice of Cleanroom. A member of the ACM and the IEEE, he is currently chair of the Boulder, Colorado, ACM chapter. He holds a BA (Mathematics) from Kalamazoo College, an MS (Computer Science) from the University of Maryland, and has taught at both institutions. Mailing address: 7526 Spring Dr, Boulder, Colorado, 80303, (303) 494-3152, deckm@cleansoft.com.

Prepared for and presented at the 1995 Pacific Northwest Software Quality Conference

# 1. Introduction

The search for new paradigms of software engineering has largely overlooked the existence of legacy systems, or "old code." CASE tools, Object-Orientation, Client-Server architectures – all of these approaches offer great value, but are primarily targeted for new systems development. "Re-Engineering" is too often a euphemism for rewriting an entire system. However, there are significant barriers to rewriting existing systems, which often place the benefits of new process improvement paradigms out of reach of many developers.

One barrier to rewriting entire systems is cost. Many of the systems that are in use today got their start almost a generation ago. There are systems in use today that were originally written in Autocoder, were automatically translated into COBOL, and have since been subjected to continuous maintenance and improvement. Such systems will probably last many more years, so the cost of a rewrite, *if amortized,* is small. But most businesses treat the cost of in-house software development as a current expense. In addition, the cost of ongoing maintenance, even if large, is a known quantity, whereas new development can be risky and unpredictable, especially when coupled with learning a new methodology. One omission from this cost/benefit analysis is the *opportunity cost* of having programmers held hostage to error correction. Instead of fixing bugs, they could be solving new problems, addressing new issues, and making a more substantial contribution to business competitiveness. This also creates a vicious cycle, since programmers and managers who are entirely engaged in firefighting have no time to learn new techniques.

The second, and perhaps more serious, barrier to rewriting an entire system is that *no one in the organization may know what the system presently does.* Documentation, which was seen as a needless expense during development, is absent or unreliable. Other risks of rewriting are huge: How many customers will be impacted? How long will it take to figure out what the system does? Even if only parts of the system need work, the interfaces and components may be so brittle with age that the choice seems always to be rewriting all of the system or none of it.

Historically, Cleanroom has also focused its attention on new systems development. As Cleanroom has matured and adapted, however, its range of applicability has grown. This paper describes Cleanroom techniques that can be used when making additions and improvements to existing systems in a variety of settings. These techniques also make it possible to introduce Cleanroom practices to a project that is already underway.

The paper begins with a brief introduction to Cleanroom principles and practices. Then, it describes some of the specification and design techniques that form the basis of Cleanroom. These techniques provide the fulcrum for a Cleanroom lever that gradually raises the quality and maintainability of existing systems. Cleanroom techniques for the old code environment are shown for specification, design, review, and testing. As with earlier works [1] the focus of this paper is on the adaptability and tailorability of the Cleanroom practices to the needs of specific projects and environments.

# 2. Cleanroom Overview

Cleanroom Software Engineering [2, 3, 4] is based on the idea that **defect prevention is better than defect removal.** It takes its name from the clean rooms used in chip manufacture. There, a scrupulously dust-free environment is maintained because it is much more expensive to remove defects from the chips after fabrication than to prevent their introduction during the process. Cleanroom has shown its merit in improving both quality and productivity in numerous settings [3, 5].

In Cleanroom, programming teams *can* and *should* strive to produce systems that are error-free upon entry to testing, so that testing can focus on certifying the reliability of the software product, rather than trying to remove as many bugs as possible from the code.

Cleanroom has three layers of meaning: Principles, which are common to *all* Cleanroom projects; Practices, which are followed by *most* Cleanroom projects; and Techniques, which *vary widely* among projects. We will describe the

principles and practices here, and then use the rest of the paper to go into some techniques that are especially suited to legacy code projects.

## 2.1 Cleanroom Principles

There are two fundamental principles that guide all Cleanroom development.

- **Design Principle**: Defect prevention in design[1] is more cost-effective than defect removal through testing.
- **Testing Principle**: The purpose of testing is to *measure* quality. Quality improvement during testing is an important side effect.

The Cleanroom principles arose out of the observation that ad-hoc, cut-and-try testing by programmers was detrimental to both quality and productivity. That activity, sometimes (perhaps erroneously) called "unit testing" or "unit debugging" was observed to lead to code bloat, "right in the small, wrong in the large" behaviors, and the proliferation of ripple effects where fixing one bug broke some other part of the program. It also gave developers and managers a false sense that something had been tested and was now "OK," only to fail in later integration and system testing.

Dr. Harlan Mills of IBM's Federal Systems Division proposed a radical change in approach, as a way to break programmers of this habit. He proposed that designers should not even compile, let alone test, any code. In this way the principles were enforced: Designers had to use thorough specifications and correctness verification in team reviews, in order to drive the total number of programming and design defects to near zero. Testing based on stochastic techniques was employed to measure Mean Time to Failure (MTTF) as a quality metric. The psychology of officially discouraging testing by designers meant that thorough code reviews had to be done prior to testing, with consequent improvements in both quality and cost-effectiveness.

Cleanroom projects have had success applying these principles, but not all of them have taken so rigorous a stand. Nearly all of them permitted compilation by designers. In addition, many projects had no way to physically prevent designer testing, and we must assume that some amount of it was done. Programmers have been very reluctant to abandon old habits, but those who have used Cleanroom rarely go back to their previous ways.

## 2.2 Cleanroom Practices

The principles alone do not define a complete methodology, but they inform and guide the selection of Cleanroom practices. The next sections summarize Cleanroom practices that are encouraged by Cleanroom advocates and that are followed, in whole or in part, by most Cleanroom teams. In every case, however, they are tailored, within the preceding principles, to fit project needs. The practices described below are grouped into three categories—management (including process), design, and testing.

### 2.2.1 Cleanroom Management Practices

Each Cleanroom project is organized into teams that take collective ownership of their work products. On large projects the teams are organized into sub-teams that reflect the system architecture. "Classic" Cleanroom divides the project team into separate design and testing teams, but even the earliest projects sometimes allowed an individual to be a member of both teams when resources were tight.

The development is *incremental*. An increment is a behavior as well as an implementation subset: Its inputs are a subset of those the completed system will accept, and it should produce outputs similar to those the completed system will produce. An increment is testable in the same environment in which the completed system will execute, a quality referred to as *end-to-end executability*. Increments are the focus of Cleanroom testing, and all

---

[1] The term "design" will include specification, design, and implementation -- all those parts of development that precede testing. The term "development" will be used to include all aspects of the software project, including management, analysis, design, and testing.

other "unit" testing is strongly discouraged. Increments accumulate into the final system through a process of continuous integration.

### 2.2.2 Cleanroom Design Practices

The Cleanroom design practices guide specification, design, implementation, and team design review. These practices apply to any software object, entity, or part. In specification, a black box view of the object is defined that hides implementation information that cannot or should not be exposed to the object's user. In design, a step is taken toward implementing the object, which integrates new (sub) specifications with a control structure (such as ifthen or whiledo) or a data structure (such as set or map). This hierarchical organization guides the team design review. Design reviews focus on both correctness (whether the design meets the specification), and design quality (whether it is efficient, simple, maintainable, and reusable.) Reviews are held frequently, and usually cover small amounts of material. Cleanroom design is not a strictly top-down process: Much designing takes place before and during the specification activity for any object. What is important is that the last intellectual pass be top-down. This forces the designers to think primarily in terms of *what* the system does, instead of *how* the system does it. This is an "elevator" model of design: One moves up and down in the hierarchy of objects, but the last ride is top-down. Specification and design practices are based on formal methods. Formal methods guide the team's choices of specification and design practices, but they may not always use the full rigor or notation of formal methods. All work products are subjected to intensive team review. Intensive team review includes everything from inspections to full formal verification. The level of rigor and formality is to some extent tied to the choice of specification and design methods.

### 2.2.3 Cleanroom Testing Practices

The primary purpose of Cleanroom *testing* is to measure and certify the reliability of the software. Software is made reliable through design, not through testing. The bulk of Cleanroom testing is black-box testing based on *usage distributions* [6] (also known as *operational profiles* [7]). Test cases are defined, chosen, or generated that reflect the expected patterns of product usage, and the success rate of the product in executing these tests is measured. The outcome of testing is a reliability prediction in terms of *Mean Time To Failure* (MTTF)[8]. Quality improvement through the correction of errors found in testing, and specification validation by demonstrating prototype increments, are also ingredients in the testing process. Again, testing by designers for the purpose of demonstrating quality is strongly discouraged.

## 2.3 Cleanroom Techniques

The practices are implemented by each project through the selection of techniques. The variety of techniques used by Cleanroom teams is large, but here are some examples:

- Testing Technique A: Operational profiles are recorded using context-free grammars, and the Certification model [8] is used to estimate MTTF.
- Testing Technique B: Operational profiles are recorded using Markov chains, and the the Markov-based models [9, 10] are used to estimate MTTF.

Each of these techniques conforms to the principles and practices, yet leaves freedom of choice for the user. Likewise:

- Design Technique A: Structured programming plus state machines
- Design Technique B: History-based box structures
- Design Technique C: Object-oriented design, guided by functional verification.

All of these, if done properly, conform to the practice of techniques based on formal methods.

Certain techniques have evolved to be considered "typical" of Cleanroom. However, Cleanroom can and should be phased-in and tailored to the specific project and organization that uses it. No team should walk away from the benefits of Cleanroom design, for example, because their testing group hasn't bought into the Cleanroom testing

part. Judgment and experience are required to guide the selection of techniques and practices that fit the overall philosophy and principles of Cleanroom and meet the needs of the project.

# 3. Specification: The Quality Improvement Fulcrum

Specification is a key element in Cleanroom. The specification serves as repository of agreement between users, designers, and testers. It documents the behavior of the system so that users can judge whether it will meet their needs, designers can verify the correctness of their designs, and testers can decide whether a particular execution succeeded or failed.

Cleanroom teams primarily rely on *behavior abstractions* for specification. A behavior abstraction emphasizes the behavior of an object while de-emphasizing the details of how that behavior is implemented. The benefits of behavior abstraction are many; chief among them is the ability to change an object's implementation without affecting or informing its users.

The *box structures* design approach [11, 12, 13] provides an organizing framework for behavior abstractions. The box structures approach plays a key role in Cleanroom [14] but it is also the foundation for work in dynamic systems modeling [15] and in Ada™ development [16]. It has three fundamental concepts:

- A *black box* is a view of an object that hides both data implementation, if any, and process implementation. We will examine two kinds of black box view, *process abstraction* and *data abstraction.* Both of these are forms of behavior abstraction.
- A *state box* is a view of an object that exposes data implementation but hides process implementation. The state box view applies to objects that encapsulate data; in other words, to those entities whose behavior abstraction is a data abstraction.
- A *clear box* is a view of an object that exposes both data implementation and process implementation. The clear box view applies to objects that hide procedurality; in other words, to those entities whose behavior abstraction is a process abstraction.

In both the state box and the clear box, the exposure of implementation details may be partial—the implementation may be expressed in a combination of *lower-level abstractions* and *primitive* (concrete) elements. This partial revelation of implementation at each level organizes the implementation into a *usage hierarchy* (sometimes called a *"uses" hierarchy* [17]). The "uses" relationship is orthogonal to the many other relationships between software entities. For example, in an object-oriented Cleanroom project, the inheritance hierarchy of "is-a" relationships will also be present. Likewise in a Clearoom development of a database system, entity relationships may also form a kind of hierarchy.

The following sections describe the three box structures in more detail. However, we will emphasize the procedure abstraction and clear box because of the larger role they play in the maintenance of legacy systems, where one is more often dealing with procedural code, and less often with large numbers of specialized data-encapsulating objects.

## 3.1 Process Abstractions

A process abstraction (also called a *procedure abstraction* [18]) emphasizes the net effect of a procedure or process on external data. It de-emphasizes procedurality and local, internal data. The notation most often used is the *conditional rule.*[19]. Figure 1 shows an example of this notation, read "if the before-value of x is greater than zero, concurrently assign the before-values of y and x to x and y respectively; otherwise, leave the values of all variables unchanged.

```
[ x > 0 → x, y := y, x
| true → identity ]
```

*Figure 1. Process Abstraction Example: Conditional Exchange*

By convention, the conditions in the conditional rule are read from top to bottom, and the assignment of the first true condition is applied. The reason for this convention is to avoid the duplication of text that would be involved in having explicitly disjoint conditions. The last condition may be written "TRUE," signifying an "else" or "otherwise" case. This notation is more fully described in Linger, et al. [19].

Underlying this notation is a functional model of programming [20]. The conditional rule expresses a program function – a mapping from domain to range – in a convenient notation. This function is a mathematical model of a process: It describes the process as if the outputs were produced all at once from the inputs, with no undocumented side effects. Other models could be used as well, including the axiomatic model [21], but the functional model is more prevalent in Cleanroom. A mathematical model is important because it permits correctness arguments to be made at any desired level of precision and rigor.

The term "process abstraction" is used here instead of the more common "intended function" because these abstractions may be *relations* – not functions in the mathematical sense – so that we can defer and circumscribe design decisions.

```
[ address_table is sorted & item is in address_table →
  return TRUE and set index to a position of item in address_table

| address_table is sorted→ return FALSE and set index to anything

| true → undefined ]
```

*Figure 2. Process Abstration Example: Table Lookup*

Figure 2 depicts a possible specification for a table lookup routine. Its first condition is satisfied if the item sought is present in the table, in which case any corresponding address is returned. If the user wanted a specific address, for example the first, a different specification would be required. In the second condition, the item sought is not in the table, so the result code is returned as FALSE and any data could be present in the address field. This is a warning to the user of the process: Don't rely on the contents if result is FALSE, and don't leave any data there that can't be overwritten. Finally, if the table is not in sorted order, the results of the lookup are *undefined.* Any behavior is possible, including system-crash, infinite loop, or erasing the database. Clearly, one goal of review will be to ensure that this can't happen. For example, this routine could always be preceded by an array sort. This kind of specification technique is very common for internal objects, whose "users" are other programs. It is less common for objects with a human user, which should handle all inputs gracefully, or for objects that will be used in a variety of environments, where the ordering of the array would not be guaranteed.

Figure 3 shows several process abstractions that operate on the same data object.

```
[ s := empty stack ]

[ size(s) ≥ 100 → result := FALSE
| TRUE → s, result := e atop s, TRUE ]

[ s = empty stack → result := FALSE
| TRUE → e, s, result := top of s, s with top removed, TRUE ]
```

*Figure 3. A Collection of Process Abstractions*

## 3.2 Data Abstractions

In Figure 1, we relied on our intuition about "pure" integers when we used the inequality and assignment operators in the process abstraction for a conditional integer swap. Similarly, in Figure 2 and Figure 3 we relied on an understanding of the statements "is sorted," "is in the array," and "the top of s." Unfortunately, our intuition is frequently wrong. It is not that assignment and equality are ill-defined in the world of pure integers, but rather that their programming-language analogues are not pure integers. In fact, what we are manipulating within process abstractions are *data abstractions.*

A data abstraction, like a process abstraction, hides and encapsulates implementation details. A data abstraction consists of a collection of *abstract attributes* (also called an *abstract model*) and a set of process abstractions – one for each *abstract method* of that object[2]. In the degenerate case, the abstract attributes and their implementation are nearly identical. In general, though, they serve different purposes. The choice of implementation depends on criteria like efficiency. The choice of abstract attributes depends instead on the *level of rigor* we desire in our correctness demonstrations, and on the *user's conceptual model* of the object's behavior. Thus, a database object would have abstract attributes like tables, rows, and cells, but would be implemented using arrays and files. The abstraction should reflect the user's view because it is the information about the object that is seen by the object's user.

```
Abstract Model
      s : stack of max 100 integer, initially empty
Abstract Methods
      initialize()
            [ s := empty stack ]
      push(e:integer) returns (result:Boolean):
            [size(s) ≥ 100 → result := FALSE
            | TRUE → s, result := e atop s, TRUE]
      pop() returns (e:integer, result:Boolean):
            [s = empty stack → e, result := anything, FALSE
            | TRUE → e,s,result := top of s, s with top removed, TRUE]
```

*Figure 4. Data Abstraction Example: Stack*

Figure 4 defines a **Stack** data abstraction. We have defined and named an abstract attribute, s, which has type "stack of integers." We have also defined a collection of process abstractions, and have named them, e.g., "pop." In the process abstractions, references to stack s are shorthand for "this.s", namely "the abstract attribute s of this instance of Stack."

Although we have chosen, for consistency, to use a functional notation for defining the data abstraction, other data-abstraction techniques [22, 23, 24] could be used. If an axiomatic approach were being used for process abstraction, one may prefer VDM [25] for data abstraction. The specification language Z [26] is of particular interest because of its combination of rigor with an abstraction hierarchy, and because its use is relatively widespread. In Z, one would build up the **Stack** definition by starting from a primitive of the specification language (such as list-of-integer) and define the **Stack** in terms of operations on this primitive. We have chosen a less formal approach and will rely on our intuition for the meaning of statements like "empty stack" and "e atop s".

[2] We have adopted some of the terminology of object-orientation, such as "method" and "attribute" to emphasize the close relatiohship between the foundations of both box structures and object-orientation. However, knowledge of object-oriented techniques is not necessary for the successful application of box structures.

The *initial value* of the stack s is important; as a data attribute, albeit abstract, it should have an initial value. In a language that permits user-defined constructors, it will be up to the constructor to initialize s.

In "classic" box structures [11, 12, 27] every data abstraction has exactly one abstract attribute, namely the object's history of use, or its *stimulus history*. The stimulus history is a list of stimuli with a most-recent and a least-recent end. Each abstract method appends a new element to this list; each has unlimited read access to the stimulus history for producing a response. In the **Stack** example, the stimulus history would contain items like **push**(i) and **pop**().

## 3.3 Systems as Abstraction Hierarchies

A specification defines an abstract model of an object. It is a black box view that can be assessed against user needs, and that can be used to determine whether or not a test was successful. A *design* (or *implementation*) is a step toward the concrete realization of a specification. There may be many possible designs for a given specification, and the designer will have to choose from among them using two criteria: *correctness* and *design quality*. A design is correct if it meets its specification. To evaluate design quality, one examines whether the design fits its purpose: is it simple, maintainable, reusable, and efficient.

### 3.3.1 Data Designs

The *data design*, or *state box*, is a collection of *state data* objects together with a collection of process abstractions, one for each of the *concrete methods*. These process abstractions are sometimes collectively called the "machine." The state data are themselves *instances* of lower-level data abstractions. The state data objects are usually *concrete attributes* (implemented as "private" or "protected" data) but they may also be intermediate abstractions that require further refinement. The state data is part of the usage hierarchy.

A state box for **Stack** is shown in Figure 5. We have chosen to have two state data objects, an array of integers and a scalar integer. In most languages, both "integer" and "array of integer" are primitives, but some languages actually implement Array as a higher-order abstraction The abstract methods have been re-expressed in terms of operations on these two data objects. The resulting process abstractions are simpler, and we now should know to take the next step toward implementing them. One aspect of "Cleanroom style" is to copy the specification on top of its implementation. We have not done this here, for reasons of space, but the example in Figure 18 shows a complete specification and implementation, in C++, of the **Stack** class using a typical style.

```
Concrete Model

      (top<0 → stack empty, else top is index of top of s)
      top : integer initial -1

      (top≥0 → a[0] is the top of s, a[top] is top of s)
      a [array 0..99] of integer;


Concrete Methods (Machine)

      initialize()
            [ top := -1 ]

      push(e:integer) returns (result:Boolean)
            [ top >= 99 -> result := FALSE
            | TRUE -> top, a[top+1], result := top+1, e, TRUE ]

      pop() returns (e:integer, result:Boolean):
            [ top = -1 -> e, result := anything, FALSE
            | TRUE -> e, top, result := a[top], top-1, TRUE ]
```

*Figure 5. Data Design (State Box) for Stack*

### 3.3.2 Process Designs

A *process design*, or *clear box*, consists of a *control structure* that coordinates *lower-level abstractions*. The basic set of control structures includes *sequence*, *alternation*, and *iteration*; most languages provide others as well.

Figure 6 shows the clear box for each of the stack methods. Some of them are trivial, such as the one-line clear box of initialize(). Others are more complex. In this example, the clear box shown would likely be expressible in primitive operations of the implementation language. However, it should be clear that any of these operations might require a call to another process, thus providing another layer to the hierarchy of abstractions. Indeed, the assignment to an array element is implemented by many compilers as a complex operation, so in a sense the statement "a[top] := e" is itself a process abstraction.

Because of its simplicity, this example does not demonstrate one aspect of the clear box, namely its ability to create local data. Naturally, each local data object is an instance of a data abstraction, whose specification is either well-known and understood, or is documented elsewhere.

```
initialize():
      top := -1

Push(e:integer) returns (result:Boolean)
      if (top >= 99) result := FALSE
      else
            top := top + 1
            a[top] := e
            result := TRUE;
      end if

Pop() returns (e:integer, result:Boolean)
      if (top = -1) result := FALSE
      else
            e := a[top]
            top := top - 1
            result := TRUE;
      end if
```

*Figure 6. Process Design (Clear Box) for each Stack operation*

### 3.3.3 Design Review

In both process design and data design, the relationship between specification and implementation is one-to-many, and there is a correctness relationship as well as design quality criteria. In Cleanroom, systematic *team review* replaces unit debugging as a way to demonstrate correctness. The benefits of *inspection* are well documented [28, 29]. Cleanroom reviews go further. They use correctness arguments based on the design's hierarchy of abstractions to guide the review, and they simultaneously examine the design for other qualities.

There are many techniques for demonstrating correctness, and considerable variation in the "standard of proof" required. To use a legal analogy, a team designing life-critical systems might ask for correctness to be demonstrated "beyond a reasonable doubt" whereas a team designing a prototype word processor might only demand that the weight of the evidence points toward correctness. Although walkthroughs and inspections may be used for this purpose, the most widely used correctness-review approach in Cleanroom is *functional verification*. Using this technique, the correctness of a process design is constructively argued using a set of *correctness questions* that are independent of the design's subject matter. Linger [19] and Dyer [4] have described techniques for verifying a process design. The techniques used to verify data designs are somewhat more varied: Mills [11] proposes a technique that is primarily useful for history-based specifications; Shankar [30] describes a more general technique.

Throughout the design process, specifications may be left undocumented. Any such omission introduces risk; it is the job of the team, during the reviews, to assess that risk and mitigate it if necessary. As the system evolves, all of

the specifications must be kept up to date to ensure reliable maintenance, and to make it easier for a new team member to understand the code. Tools are being developed [31] that will assist the designer in the management of specification and design libraries.

# 4. Specification, Design and Review for "Old Code"

If the essence of Cleanroom development is specification, its absence is the most serious problem facing Cleanroom developers in the legacy systems environment. Because Cleanroom design teams maintain intellectual control by organizing the system as a hierarchy of specifications, and by reviewing correctness relationships in that hierarchy, any holes in that hierarchy would seem to make the structure collapse and prevent any use of Cleanroom at all. Because the majority of programming work is focused on maintaining legacy systems, however, some specialized techniques must be used in Cleanroom in order to make that specification hierarchy whole enough to be useful. The use of Cleanroom techniques in legacy systems maintanance addresses many of the problems inherent in that maintenance, including:

- understanding and using interfaces with old code,
- estimating the impact of a change,
- knowing which old components can be re-used, and
- avoiding major negative impact to current users.

The next sections will describe a few of the specification, design, and verification techniques that are especially useful for Cleanroom teams operating in a legacy code environment. The application of the techniques should of course be done within the framework of the principles and practices described above.

We group together specification and design because, as we have seen, a "design" is a judicious decomposition of the specification into sub-specifications linked together with control and data structures.

```
[new behavior abstraction of system]
procedure old_system_improved

      [behavior abstraction of old component a]
      call component a

      [behavior abstraction of new component x]
      call component x

      [behavior abstraction of old component b]
      call component b

end procedure
```

*Figure 7. "Maintenance" Scenario Example*

We will focus primarily on two scenarios in which a team is using Cleanroom techniques to work with legacy code. The first, which we will call "maintenance," involves the team adding a relatively small amount of new code or new functionality to an existing system. The second code scenario we will call "rewrite," though it can apply both to system parts as well as to entire systems. Using pseudocode, Figure 7 shows how the new overall system behavior has been changed by the insertion of a call to new component x in between calls to original components a and b. This would be the case when a new function or feature, comprising from a few lines to many thousands of lines, is added to a large base of existing code. Figure 8 shows how a new overall system or part has been constructed by writing mostly new code (encapsulated in new components a and b) but calling upon functionality from the old system in the form of component x. This would be the case when a relatively independent new function is added to an existing system, or when an existing system is overhauled and rewritten.

```
[behavior abstraction of new system or part]
procedure new_system_part

      [behavior abstraction of new component a]
      call component a

      [behavior abstraction of old component x]
      call component x

      [behavior abstraction of new component b]
      call component b

end procedure
```

*Figure 8. "Re-use" Scenario Example*

At issue in either scenario is the mixing of old code that is poorly understood, and new code that is being developed using Cleanroom practices In the maintenance scenario, we would like the new behavior abstraction to reflect reality. We know the behavior of x, but we don't know the behaviors of a and b, so abstraction and verification are difficult. In the rewriting scenario, the call to old x amid new code again makes it difficult to verify the new overall behavior.

There are several possible solutions that will be described in the next sections. The section entitled **Specification Recovery** describes the most rigorous approach, which is to take the old code that is being used and derive its behavior abstraction. This is a very complex, difficult, and time-consuming task, but one that has great merit if the old code is either very important or is expected to be long-lived.

**Specification Interposition** describes a second alternative, which is to assume for each old code part whatever behavior abstraction is needed to verify the desired overall behavior. This behavior abstraction then becomes part of the requirements for the old code part, and we have decomposed our problem into verifying the overall behavior based on our assumption, and arguing that each old component has that assumed behavior. The latter can be done through rigorous specification recovery, through interface experimentation (i.e., running test cases to probe the old code), through discussion with the developers or maintainers of the old code, or a combination of these methods.

**Relative Specifications** describes a third alternative, which is to expose the existence of the component to the user, in effect making the new behavior "relative" to the old code. This may be the simplest path, and is attractive when the "user" is another computer or is another department of the same organization, but is less useful for an end-user or customer.

Many of these techniques are used informally today by programming teams that are not using the Cleanroom approach. The goal of this section is to show how these techniques, properly managed, can permit legacy systems maintainers access to the full range of Cleanroom benefits.

For reasons of space, we will use a simple example to illustrate the various techniques. We have chosen a "reuse" example, in which a new procedure uses existing services to carry out a new function. In the example, shown in Figure 9, the calls to "old" parts are shown in bold face type. All of these techniques apply to the maintenance scenario with only minor adaptations.

```
[if the user enters a database key on form GetDatabaseKey that is invalid (see
… for validity checks), display a message appropriate to the validity problem.
Otherwise display the database record for key on form ShowData.]

procedure NewDatabaseLookup

      [display form GetDatabaseKey for user to enter key]
      call Form.GetDatabaseKey(key)

      [unknown]
      call DB_Server.Lookup (key, result)

      if (result ≠ OK)

            [display result in a message box]
            call MessageBox(result)

      else

            [unknown]
            call DB_Server.GetData(key, data);

            [display database record data on form ShowData]
            call Form.ShowData(data);

      end if
end procedure
```

*Figure 9. NewDatabaseLookup*

## 4.1 Specification Recovery

The details of specification recovery are thoroughly covered in [19, 32]. Our goal here is to briefly summarize the process, and to present some of the practical issues as they relate to the old code problem.

Suppose we have a sequence such as that shown in Figure 10, and we wish to abstract its overall behavior. Note that we're already beginning part-way "up" in the design hierarchy, with some behavior abstractions for the table lookup, flag checking, and buffer validation.

```
procedure SetItemMask (in item, map_tbl; out msk)

      [item is in map_tbl → msk1 := x'80'
      | TRUE → msk1 := x'00']
      call MapTableLookup (in item, map_tbl; out msk1)

      [item has flag on → msk2 := x'40'
      | TRUE → msk2 := x'00']

      [item buffers are OK → msk3 := x'20'
      | TRUE -> msk3 := x'00']

      msk = msk1 OR msk2 OR msk3;

end procedure
```

*Figure 10. SetItemMask Sequence*

A common technique for abstracting a sequence such as this is the trace table. For each step, we summarize the net effect so far on all the data, expressing that net effect in terms of functions on the input variables. This process is shown in Figure 11.

| Step | Cumulative Behavior |
|---|---|
| [item is in map_tbl → msk1:=x'80'<br>\| TRUE → msk1:=x'00'] | [item is in map_tbl → msk1:=x'80'<br>\|TRUE → msk1:=x'00'] |
| [item has flag on → msk2:=x'40'<br>\| TRUE → msk2:=x'00'] | [item is in map_tbl &<br>item has flag on→<br>msk1:=x'80', msk2:=x'40'<br>\|item is in map_tbl &<br>item has flag off→<br>msk1:=x'80', msk2:=x'00'<br>\|item is not in map_tbl &<br>item has flag on→<br>msk1:=x'00', msk2:=x'40'<br>\|TRUE → msk1:=x'00', msk2:=x'00'] |
| [item buffers are OK → msk3:=x'20'<br>\| TRUE -> msk3:=x'00'] | [item is in map_tbl & item has flag on &<br>item buffers are OK →<br>msk1:=x'80', msk2:=x'40', msk3:=x'20'<br>[item is in map_tbl & item has flag on &<br>item buffers are not OK →<br>msk1:=x'80', msk2:=x'40', msk3:=x'00'<br>\|item is in map_tbl & item has flag off &<br>item buffers are OK →<br>msk1:=x'80', msk2:=x'00', msk3:=x'20'<br>\|... (4 cases elided)<br>\|TRUE → msk1:=x'00', msk2:=x'00', msk3:=x'00'] |
| msk = msk1 OR msk2 OR msk3; | [item is in map_tbl & item has flag on &<br>item buffers are OK →<br>msk:=x'E0',<br>msk1:=x'80', msk2:=x'40', msk3:=x'20'<br>\|... (6 cases elided)<br>\|TRUE →<br>msk:=x'00',<br>msk1:=x'00', msk2:=x'00', msk3:=x'00'] |

*Figure 11. Trace-Table Analysis of SetItemMask*

Unfortunately, the mechanical abstraction of this example results in a "case explosion" of 8 cases. What's required here is the application of some intelligence, to know why this work is being done, and (if it isn't clear from the language) whether the final value of each variable is important or not.

One of the techniques we can use to record this intelligence is specification functions, or single-valued functional abstractions [33], which give us a way to name a mathematical function on one or more arguments. These specification functions do not necessarily map to functions or procedures in the code, but are artifacts of our understanding. In the preceding example, suppose we know (from domain or from context) that:

- the independent values of flag-on and buffers-OK are a remnant of prior work -- meriting independent tests but no longer significant
- the unique responses for combinations of flags and buffers is immaterial to the user
- the values of msk1, msk2, and msk3 are local to this procedure

So, we might write a function called "FlagAndBuffersOK" defined as in Figure 12. The resulting overall function becomes considerably simpler, as shown in Figure 13.

```
defn FlagAndBuffersOk (item) : byte is
  x'60' if item flag is on and buffers OK
  x'40', x'20', or x'00' otherwise.
end defn
```

*Figure 12. Specification Function FlagAndBuffersOK*

```
[item is in map_tbl →
  msk := x'80' AND FlagAndBuffersOk(item)
|TRUE →
  msk := x'00' AND FlagAndBuffersOk(item)]
```

*Figure 13. SetItemMask Final Abstract Behavior*

The domain and program knowledge needed to collect fragmentary information into coherent units of understanding is key to any success at specification recovery.

Unfortunately, one reason for doing specification recovery is that such program and domain knowledge is absent, since with that knowledge the Specification Interposition method would clearly be more appealing. Further, in a case such as the example in Figure 9, the source code for the unknown parts may be unavailable. Nevertheless, though pure, 100% bottom-up specification recovery is extremely tedious, difficult, and time-consuming, judicious combination of it with the Specification Interposition approach can be very useful in providing insight and understanding.

## 4.2 Specification Interposition

In contrast to the pure "bottom-up" approach of Specification Recovery, the Specification Interposition approach is more of a "middle-out" technique. It calls for us to interpose an "assumed" behavior abstraction for each unknown part, and then work in two directions: verify the overall behavior given the assumed part behaviors, and argue that the parts do carry out their assumed behaviors. In the example of Figure 9 (NewDatabaseLookup) we had two unknown parts. Either by guesswork, or experience, or trial and error, we arrive at a behavior specification for each part that permits us to verify that the program has the desired overall behavior. These are shown in Figure 14.

| Component | Behavior Abstraction |
|---|---|
| DB_Server.Lookup (key, result) | `[key is valid (see ... for validity checks)→result := OK`<br>`\| true → result := not OK ]` |
| DB_Server.GetData (key, data) | `[key is not valid (see ... for validity checks)→undefined`<br>`\| true → data := database record for key ]` |

*Figure 14. Interposed Specifications for NewDatabaseLookup*

We can plug these abstractions into the original problem and derive an overall behavior using trace table analysis. The result is shown in Figure 15. Note that, because the Lookup guards the GetData, we don't have to carry the undefined case out to the user.

```
Desired:
[if the user enters a database key on form GetDatabaseKey that is invalid (see
... for validity checks), display a message appropriate to the validity
problem. Otherwise display the database record for key on form ShowData.]
Derived:
[if the user enters a database key on form GetDatabaseKey that is invalid (see
... for validity checks), display a message appropriate to the validity
problem. Otherwise display the database record for key on form ShowData.]
procedure NewDatabaseLookup
      [display form GetDatabaseKey for user to enter key]
      call Form.GetDatabaseKey(key)
      [key is valid (see ... for validity checks)→resp := OK
      | true → result := not OK ]
      call DB_Server.Lookup (key, result)
      if (result ≠ OK)
            [display result in a message box]
            call MessageBox(result)
      else
            [key is not valid (see ... for validity checks)→undefined
            | true → data := database record for key ]
            call DB_Server.GetData(key, data);
            [display database record data on form ShowData]
            call Form.ShowData(data);
      end if
end procedure
```

*Figure 15. NewDatabaseLookup with Interposed Specifications*

The result is a perfect match! However, if it had not been, the differences between the derived function and the desired one would guide us to a better interposed function for either or both of the unknowns, until we arrived at a correct design assuming the interposed functions are correct.

The next step would be to convince ourselves that the two DB_Server functions will meet our expectations. Unfortunately, the ideal way to do this is by specification recovery, preceding. That technique avoids missing any important behaviors, but it requires that the source code be available, it requires considerable resources, and it presents the case explosion problem whose minimization requires human intelligence and hence introduces the possibility of error.

One alternative to specification recovery is experimentation. This may be done in an ad-hoc fashion by trying a few inputs and seeing the results, or it may be done more systematically. Black box techniques are frequently used—testing random input patterns, testing patterns based on expected usage, or testing "boundary" conditions. If the source code is available, thorough "unit" testing combined with a coverage analyzer can identify a fair amount of the function, and specification recovery can fill in the gaps.

## 4.3 Relative Specifications

Our third strategy introduces the idea of "relative" specifications. Suppose we know nothing at all about these two DB_Server functions except that key is unchanged. We can at the very least write down behavior abstractions such as those shown in Figure 16.

| Component | Behavior Abstraction |
|---|---|
| DB_Server.Lookup (key,result) | result := whatever value DB_Server.Lookup returns for argument "key" |
| DB_Server.GetData (key, data) | data := whatever information DB_Server.GetData returns for "key" |

*Figure 16. Relative Specifications for NewDatabaseLookup*

The result of using our abstraction techniques to derive the overall behavior in this case is shown in Figure 17.

```
Desired:
[if the user enters a database key on form GetDatabaseKey that is invalid (see
... for validity checks), display a message appropriate to the validity
problem. Otherwise display the database record for key on form ShowData.]
Derived:
[if the user enters a database key on form GetDatabaseKey for which
DB_Server.Lookup returns anything but OK, display a message containing
whatever value DB_Server.Lookup returns for the entered key. Otherwise display
whatever information DB_Server.GetData returns for that key on form ShowData.]
procedure NewDatabaseLookup
      [display form GetDatabaseKey for user to enter key]
      call Form.GetDatabaseKey(key);
      [result := whatever value DB_Server.Lookup returns for argument "key"]
      call DB_Server.Lookup (key, result)
      if (result ≠ OK)
            [display result in a message box]
            call MessageBox(result)
      else
            [data := whatever information DB_Server.GetData returns for "key"]
            call DB_Server.GetData(key, data);
            [display database record data on form ShowData]
            call Form.ShowData(data);
      end if
end procedure
```

*Figure 17. Relative Specifications applied to NewDatabaseLookup*

The result is cumbersome, true, but maintains the arms-length relationship between the new code and the old. A tester of this function could monitor calls to the DB_Server routine, and could use this specification to objectively determine whether a test case had succeeded or failed. Further, if DB_Server ever changes, there is no impact to the new behavior – it still just parrots back the responses from DB_Server.

Of course, there are significant drawbacks to this approach as well. First and foremost is that an end-user would be dismayed to find a description like the derived abstraction in a manual! The user doesn't care about the interfaces or the relative roles of the two parts, they wanted the original request! And we've hidden ourselves so well behind the shield of the relative specification that it's impossible to make any kind of case that the derived function actually meets the requirements, without drawing on additional knowledge about DB_Server.

The technique of relative specifications is especially useful in the "maintenance" scenario, where a small fraction of the system's functionality might be changed. There, we might see a specification that says, "for inputs A, B, and

C, the resulting output is exactly what the prior release produced; for input D, the output is (some new function on inputs)." Again, this has the drawback of sending the reader off to another document to find the behavior specification for most of the domain, but in an environment where the specification on A, B, and C is not, or cannot be written down, it is an important alternative.

Making a specification "relative" to other documents makes some things easier and other things harder. By weighing the importance of these issues, and the costs to achieve complete independence, a team can arrive at a prudent strategy.

# 5. Testing Techniques for "Old Code"

This section briefly examines ways for tailoring the software reliability measurement techniques used in Cleanroom testing to address the special issues of new code mixed with old.

## *5.1 Statistical Quality Control for Software*

Traditionally, Cleanroom testing has focused primarily on predicting the Mean Time to Failure(MTTF) under expected usage conditions. This user-centered view of quality is a significant departure from designer-centered metrics such as defects per unit of code, and it arises from the Cleanroom principle that the purpose of testing is not to remove defects but to measure quality.

Most Cleanroom testing begins with an understanding of how the system is to be used, with this understanding being recorded in an *operational profile* or *usage distribution.* The operational profile defines a probability for every possible usage scenario. This probably distribution defines a sampling technique wherein the universe of all possible customer usage scenarios is described. By selecting scenarios based on this probability distribution, the techniques of statistical quality control can be applied to software: A scenario is drawn from this distribution, executed by the system, objective measurement of success or failure is provided, and the contribution of that test to the overall quality profile is determined. There are several models typically used in Clearoom to infer MTTF. The Certification model [8] is the reliability growth model originally proposed for use with Cleanroom. Markov-based models [9] avoid some of the problems of reliability-growth models [34] and are gaining somewhat in favor among the Cleanroom community.

It is a significant engineering task to understand and record the operational profile for a product, and one that is best done incrementally along with the product's development. However, in a legacy code environment, one will rarely find an operational profile as part of the system's documentation. Just as rewriting the entire system would be prohibitively expensive, so would be creating a complete operational profile from scratch.

In the case where we are writing a new end-user function that re-uses parts of the old code, we can treat that function as a system, with its own distribution of inputs. We can use statistical testing based on that distribution to arrive a MTTF for *that function.* We cannot deduce the MTTF of the overall system from this, but we can use it to guide process-improvement efforts, and we can use it to estimate maintenance costs for that new function. This techique becomes more difficult, the more intertwined the new function is with the old, but it can have significant benefits especially in learning how to carry out statistical testing. This technique can also be used in the case where we are adding a new behavior to an existing system. We can determine the subset of the overall product's operational profile that is affected by the new behavior, and we can focus our sampling on that subset. We arrive at a MTTF that relates only to that subset, but over time the size and validity of the profile will grow and the product will be correspondingly improved.

The preceding is an example of a flexible approach used in Cleanroom testing. Although still predominantly black-box in nature, by stressing certain functions based on something other than expected usage we are departing from the strict operational profile view of "orthodox" Cleanroom testing. Many teams have used this idea of "target profiles," or stress testing, where the probability of some subset of the sample space is artificially elevated in order to focus testing effort on a system part or function. This technique is also done in new systems development, when a low-likelihood subset of the sample is critical to the user, and may have a much higher required MTTF. Voas and Miller 35 have suggested using testability analysis as a way to guide both verification rigor and targeted testing.

Depending on how we define "time" and "failure," MTTF is a good or bad predictor of user satisfaction. In control systems and the like, failures are defined objectively by a specification, and time is usually measured in units like CPU hours or clock time. In applications, however, failure is more subjective, depending not only on whether the program meets its specification, but also whether the specified behavior meets user expectations. A further complication is that failures to meet expectations may not have a defined point in time. For example, if a user finds the interface "cumbersome" at what point did it fail? Cleanroom teams that have developed applications software have, in the author's experience, augmented strict reliability-based testing with other kinds of black box testing including usability testing, claims testing, and other techniques in order to ensure commercial success in addition to meeting reliability goals.

## 5.2 Use of Oracles

In the preceding section entitled Relative Specifications, we looked at how a new system's specification (whether maintenance or reuse) could be defined relative to another system or collection or parts. Certainly when we have an unspecified system V1, and we are adding a small delta to it resulting in V2, whether or not the behavior of V1 is well-documented we should consider using V1 as an oracle for deciding whether V1 test cases succeeded or failed. To do this, we must separate our test case distribution into those that exercise only V1 features, and those that exercise V2 features (or both V1 and V2 features). All the test cases in the former category can be supplied to both V1 and V2, and the results compared. The test cases in the latter category must have their outcomes checked against the V2 specification. Where a reuse type of specification is relative (e.g., "display the result of calling function X with key Y") the interface may need to be instrumented in order to check the success or failure of a test case.

## 5.3 Regression Testing

Historically, the Cleanroom testing principle has led Cleanroom teams to eschew most regression testing. The reasoning goes as follows. Regression testing is usually done for two reasons: to demonstrate that we didn't introduce new errors in the original function during a maintenance operation, and to demonstrate that we fixed what we were trying to fix. Now, suppose we did introduce new errors. Then, by testing based on operational profiles, those errors would at some point contribute to a failure which would be seen and corrected, and we would in addition have an MTTF for the maintained system. And, if the failure we were trying to correct occurred once, it should occur again (assuming our operational profile is accurate). If we didn't actually correct the problem, we will see another instance of that failure and will have another opportunity to correct it. In both cases, we must remember that it is not the job of testing either to ensure the absence of unintended side effects caused by the maintenance, or to ensure the adequacy of that maintenance: that is the job of the design team.

In the case of using Cleanroom for old code, though, we may need to relax this posture a bit. For one thing, placing all of the responsibility for quality on the design team is a bit onerous, since they may be dealing with a large base of code that is poorly documented or understood. For another thing, we may not have had the resources to develop a complete operational profile for the entire system, including all that which pre-dated Cleanroom. So we may not have test cases that reflect a profile-based testing scheme. For these reasons it may be prudent to re-use prior (non-Cleanroom) test cases, and to do considerable regression testing (using the prior version as an oracle when appropriate). The literature on regression testing, and its application to maintenance, is extensive. Beizer [36, 37] may be the most accessible and thorough representative of that literature.

Another reason for avoiding regression testing is that failure data gleaned from sources other than operational profiles, including the re-use of prior test buckets or from regression testing, is of no use in estimating MTTF. So, regression testing is best used *after* a baseline MTTF has been determined using the statistical techniques.

# 6. Conclusion

This paper has described some ways that Cleanroom techniques and practices can be extended to make Cleanroom accessible to maintainers of "old code." On the specification, design, and implementation side of Cleanroom, these techniques focus on ways to recover or define specifications so as to fill in gaps in the information hierarchy that

permits design review. The specification and review techniques are applicable both in a "maintenance" scenario and in a "re-write" scenario. On the testing side, we have suggested that "orthodox" Cleanroom testing techniques may require relaxation, including the use of regression testing, when systems containing substantial amounts of "old code" are tested.

Because these techniques can be applied to a greater or lesser degree, and because they are useful in dealing with a body of existing code, they are a natural way to introduce Cleanroom practices after a development project has already begun. This overcomes yet another process improvement barrier: that there is no good time to introduce a methodology change because projects tend to overlap. By gradually introducing the Cleanroom techniques for old code, an ongoing project can become comfortable with using Cleanroom and will be ready to embrace it more fully when the next development cycle begins.

A comparison of this paper with the first writings on Cleanroom indicates just how far Cleanroom has come, and how flexible it is. Now, it is true that the "Cleanroom community" is currently debating and discussing the direction of the methodology. And, there are still some voices in that community that advocate a rigid orthodoxy. But this situation is nothing new in methodology, and is in fact strongly reminiscent of the "religious debates" in the object-oriented community not that long ago. Fortunately, as more teams use Cleanroom and *report on their results*, the tide is certain to shift, and Cleanroom will follow object-orientation in the path of helping software practitioners to be more successful and productive.

## Acknowledgements

The author would like to thank the anonymous reviewers for their many helpful suggestions. A large number of individuals and organizations downloaded the preliminary draft from the Internet, and their comments are much appreciated. A special note of thanks is due Dr. Boris Beizer for raising several important issues. Although time and space did not permit responding to all of his comments here, they will be useful in the author's future work.

# 7. Appendix: C++ Example

The following example combines the data abstraction (black box), data design (state box), and process designs (clear boxes) for the integer Stack, using C++ and a "typical" Cleanroom style of documentation.

```
class Stack {
 //    s : stack of integer
public:
//      [ s := empty stack ]
        Stack() ;

//      [ s := anything ]
        -Stack() ;

//      [ s := empty stack ]
        void initialize() ;

//      [ size(s) >= 100 -> return FALSE
//      | TRUE -> s, return := e atop s, TRUE ]
        BOOL push (int e) ;

//      [ s = empty stack -> return FALSE
//      | TRUE -> e, s, return := top of s, s w/ top removed, TRUE ]
        BOOL pop (int& e)

protected:
        int a[100];
        int top;
};

// [ top := -1 ]
Stack::Stack()
        { top = -1; }

// [ top, a := anything in 0..99, anything ]
Stack::~Stack();
        {}

// [ top := -1 ]
void Stack::initialize()
        { top = -1; }

// [ top >= 99 -> return FALSE
// | TRUE -> top, a[top+1], return := top+1, e, TRUE ]
BOOL Stack::Push(int e)
        { if (top >= 99) return FALSE;
          else
                { a[++top] = e;
                  return TRUE;
                }
        }

// [ top = -1 -> return FALSE
// | TRUE -> e, top, return := a[top], top-1, TRUE ]

BOOL Stack::Pop(int& e)
        { if (top == -1) return FALSE
          else
                { e = a[top--];
                  return TRUE;
                }
        }
```

*Figure 18. Stack Example in C++*

# 8. References


1 Deck, M.D., "Cleanroom Software Engineering: Quality Improvement and Cost Reduction," 1994 Pacific Northwest Software Quality Conference, Octiber, 1994, pp. 243-258.

2 Mills, H.D., M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software*, September, 1987, pp.19-25.

3 Cobb, R.H., and H.D. Mills, "Engineering software under Statistical Quality Control," *IEEE Software*, November, 1990, pp. 44-54.

4 Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.

5 Linger, Richard C., "Cleanroom Software Engineering for Zero-Defect Software", *Proc. 15th International Conference on Software Engineering*, May, 1993.

6 Poore, J.H., Mills, H.D., and D. Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software*, January, 1993, pp. 88-99.

7 Musa, J.D., "Operational Profiles in Software-Reliability Engineering," *IEEE Software*, March, 1993, pp. 14-32.

8 Curritt, P.A., M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, (January, 1986), pp. 3-11.

9 Whittaker, J.A., and Poore, J.H., "Statistical Testing for Cleanroom Software Engineering," Proc. 25th Hawaii International Conference on System Sciences, January, 1992.

10 Whittaker, J.A., and Thomason, M.G., "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, October, 1994, pp. 812-824.

11 Mills, H.D., R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, 1985.

12 Mills, H.D., R.C. Linger, and A.R. Hevner, "Box Structured Information Systems," *IBM Systems Journal*, vol. 26, no. 4, pp. 395-413, 1987.

13 Mills, H.D., "Stepwise Refinement and Verification in Box-Structured Systems," *IEEE Computer*, June, 1988, pp. 23-36.

14 Hevner, A.R., S.A. Becker, and L.B. Pedowitz, "Integrated CASE for Cleanroom Development," *IEEE Software*, March, 1992, pp. 69-76.

15 Becker, S.A., and A.R. Hevner, "A Dynamic System Modelling Tool using Box Structures and Petri-Nets," *Conference on Dynamic Modeling of Information Systems*, July, 1991.

16 Donaldson, C.M., E.R. Comer, and A. Rudmik, "Ada Box Structures: Starting with Objects," *Proc. Washington Ada Symposium*, June, 1990, 123-132.

17 Parnas, D.L., "On a 'Buzzword': Hierarchical Structure," *IFIP Congress 74*, North Holland Publishing Company, 1974, pp. 336-339.

18 Shankar, K.S., "Data Structures, Types, and Abstractions," *IEEE Computer*, April, 1990, pp. 67-77.

19 Linger, R.C., H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Reading, MA: Addison-Wesley, 1979.

20 Mills, H.D., "The New Math of Computer Programming," *Communications of the ACM*, vol. 18, no. 1 (January, 1975), pp. 43-48.

21 Hoare, C.A.R., "Algebra and Models," *Proc. 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December, 1993. (also published as *Software Engineering Notes*, vol. 18, no. 5, December, 1993, D. Notkin, ed., pp. 1-8).

22 Liskov, B., and S. Zilles, "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 3 (March,1975), pp. 7-19.

23 Wulf, W.A., R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4 (December,1976), pp. 253-265.

24 Guttag, J.V., E. Horowitz, and D.R. Musser, "Abstract Data Types and Software Validation," *Communications of the ACM*, vol. 21, no. 12, (December,1978), pp. 1048-1063.

25 Jones, C.B., *Systematic Software Development using VDM*, second edition, Prentice-Hall, 1990.

26 Wordsworth, J.B., *Software Development with Z*, Addison-Wesley, 1992.

27 Deck, M.D., M.G. Pleszkoch, R.C. Linger, and H.D. Mills, "Extended Semantics for Box Structures," *Proc. 25th Hawaii International Conference on System Sciences*, 1992, pp. 382-393.

28 Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3 (1976) pp. 219-249.

29 Weller, F., "Lessons from Three Years of Inspection Data," *IEEE Software*, September, 1993, pp. 30-45.

30 Shankar, K.S., "A Functional Approach to Module Verification," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 2 (March, 1982), pp. 147-160.

31. Fuhrer, D., H. Mao, and J.H. Poore, "OS/2 Cleanroom Environment: A Progress Report on a Cleanroom Tools Development Project", *Proc. 25th Hawaii International Conference on System Sciences*, 1992, pp. 449-458.

32 Hausler, P.A., M.G. Pleszkoch, R.C. Linger, and A.R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, January, 1990.

33 Pleszkoch, M.G., P.A. Hausler, A.R. Hevner, and R.C. Linger, "Function-Theoretic Principles of Program Understanding," *Proc. 23rd Annual Hawaii Conference on System Sciences*, January, 1990.

34 Fenton, N.E., *Software Metrics: A Rigorous Approach*, Chapman and Hall, London:1991.

35 Voas, J.M., and K.W. Miller, "Software Testability: The New Verification," IEEE Software, May, 1995, pp. 17-28.

36 Beizer, B., *Software Testing Techniques*, 2d ed., Van Nostrand Reinhold, 1990.

37 Beizer, B., *Black Box Testing*, John Wiley and Sons, 1995.

# THE CLEANROOM PROCESS MODEL: A CRITICAL EXAMINATION

**Address at Pacific Northwest Quality Conference**
**Portland Oregon, September 28-29, 1995**

**Prepared By:**
Boris Beizer, PhD
ANALYSIS
1232 Glenbrook Road
Huntingdon Valley, PA 19006
PHONE: 215-572-5580
FAX : 215-886-0144
Email: BBEIZER@MCIMAIL.COM

Prepared For:
File

## 1. Setting The Ground Rules

### 1.1. Historical Perspective

Since its inception in 1980 as a proposed software development methodology Cleanroom has enjoyed considerable publicity. As with many ideas that seem good at first, such as Harlan Mills' previous Chief Programmer Team methodology [BAKE94], upon deeper examination they appear to work, but for the wrong reasons. At the time (1969-1972), the Chief Programmer team promised a solution to ***The Software Problem***—too many bugs that are too expensive to find and fix. Later attempts at using this method (even within IBM) showed that it did not transfer to other projects and it did not scale up [BAKE94]: it worked only if you had the right chief programmer. The retrospective consensus was that the right chief programmer could probably have achieved success with any process model. Yet, much of value was learned from the experience that earned a place in the mainstream of software development: structured programming, top-down design, egoless programming, configuration control, walkthroughs. While these ideas did not originate with Mills, he was a pioneer in their use. The contribution of the Chief Programmer Team was valuable even though its central thesis was flawed; even though why it first appeared to work was not well understood.

### 1.2. Previous Promised Panaceas—The Thirty Month Locusts

The Chief Programmer Team is only one of many panaceas that were offered to revolutionize software development. A new one emerges every 30 months. Some of these we wish we hadn't dreamed of: others have earned their portion within an ever-improving software development process. What are some of the previous formulas that promised to solve ***The Software Problem***?

1. COBOL. Read the early promises of the COBOL committee in the 50's and how they believed that this HOL would solve ***The Software Problem.***
2. Documentation
3. Stubs
4. Decision Tables
5. Modularity
6. Structured Programming.
7. Walkthroughs, Reviews, and Audits
8. Functional Decomposition
9. Symbolic Execution
10. Formal Proofs of Correctness
11. Multiple, Independent Programming
12. Formal Specification Languages
13. Software Factories
14. Top-Down Design
15. CASE
16. Ada
17. Cleanroom
18. Object-Oriented Programming
19. TQM

I left out formal (i.e., Fagan, FAGA76) inspections, unit and integration testing, configuration control, prototyping, joint requirements development, data flow design, incremental development, metrics, and other staples of good processes because even their most passionate advocates do not claim them to be panacean. Noteworthy among the missing are proprietary methodologies and software representation schemes that have found broad use such as Bender's, Softech, HIPO, Jackson, Structure Charts, Yourdon, and others. I call these "proprietary" even though in some cases they are not sold as products as such: they are "proprietary" because they are based on personal heuristics rather than on fundamental science. I am not opposed to heuristics or ad-hoc, personal methods, if they are empirically demonstrated to be of value in practice: just don't confuse them with science.

The above items have been, and are still promised by some, as the total, singular, solutions to ***The Software Problem***. A half-century of software development has taught us that there is no singular solution to ***The Software Problem***. An eloquent statement of that thesis is Fred Brooks' *No Silver Bullet* paper [BROO86]. These proposed methodologies come out of the woodwork every 30 months and have done so from the beginning of the industry. The first sixteen items on the list are rarely claimed today as the panacea they were once thought to be because working with them has revealed what they can and can't do: they have, to the extent warranted, been incorporated into the best standard process. Object Oriented Programming and TQM (as panacea, that is) are rapidly fraying at the edges; but as always, the valued parts are and will continue to become embedded in the best standard process.

### 1.3. Which "Cleanroom"—Orthodox versus Liberal?

As a leading Cleanroom advocate, Michael Deck, pointed out to me [DECK95] and as implied in his survey of Cleanroom [DECK94], Cleanroom is not monolithic. There is considerable variability in what is or is not included under the Cleanroom umbrella. Some advocates, such as Mills [COBB90, DYER92, HAUS94, LING94, MILL94] insist that programmers not compile their own code—while others allow it. Most condemn "unit testing", while others permit it under special circumstances and for "experimental purposes" [DECK94, SELB87]. So Cleanroom is not a monolith. Indeed, as Deck [DECK94] puts it: "It is important to observe that few of the individual ideas, techniques, or tools in Cleanroom are or were revolutionary. Cleanroom has always incorporated many of the best ideas of Computer Science."

Is "Cleanroom" then, just another collective term for a good process, or is it something different? Cleanroom differs in several fundamental ways from the best standard process, as will be shown below. The most important difference is the universal condemnation by Cleanroom advocates of "unit testing" and indeed, of most "test techniques." I put quotes around unit testing and test techniques above because it is clear from the Cleanroom literature that they mean very different things by these terms than the standard usage implies.

What is there to criticize about Cleanroom if the process in question is open to such great variability so that in some versions, it is indistinguishable from a good standard process? If the definition shifts, then attempting a criticism, as I have learned, is like trying to nail Jello to a wall. Accordingly, in order to establish a fixed benchmark, I will criticize what I call "Orthodox Cleanroom": a concept clearly defined by Harlan Mills. I used the following recent sources for that definition: (1) The article by Harlan Mills in the Encyclopedia of Software Engineering [MILL94], (2) the article by Richard C. Linger [LING94] in IEEE Software, and (3) the book by Mike Dyer [DYER92]. I call the process described in these sources "Orthodox Cleanroom." Why "Orthodox?" Because, although the Cleanroom advocates admit that in practice, the true, ultimate Cleanroom Process has not been achieved, it is the goal to which they aspire [HAUS94]. If those goals, however Utopian they be, are dangerous (as I believe) then whether or not they have been achieved in practice, they must be criticized. Conversely, if these espoused goals are not true intentions, but merely motivational rhetoric, then espousing them is no better than propaganda—in supposedly scientific literature, people say what they mean and mean what they say and do not excuse their exceptions to these fundamental principles of scientific discourse by claiming "it's only rhetoric."

### 1.4. Making Fair Comparisons

The earliest papers by Mills on Cleanroom were heavy in their condemnation of what was then (late 70's) an admittedly awful software development process. This seems to have established a precedent for Cleanroom literature because almost every paper perpetuates this tradition.[1] Is Cleanroom an improvement over a 1970's process? Of course it is; almost anything would be; but the software development process has evolved since 1975. If we are to make a fair comparison, then it should be Cleanroom versus software development at its best in 1995 and not to software development at its worst in 1975. I've used the term "best standard software development process" above, and will use it below, to denote that evolved process. I don't mean the most commonly used process because unfortunately, 75% of all software development is still done under a process (if you can call it that) that has hardly changed since 1955. "The best standard process" is such in the sense that it reflects the best current thinking and practices in software development, and not "standard" in the sense that it is in dominant use.

There is no single book or article that describes that process, although some books, such as BLUM92, come close. I used to write about the software development process when I thought that I understood it [BEIZ84]. The more I learned, the more I realized that it was almost impossible to describe the best process because of its awesome and necessary diversity. We can't write books on "**The Correct Way**" to do architecture: we can only define the options and list the factors that indicate the use of this

1. It is instructive to note that in one of Mills' latest discussions of Cleanroom [MILL94], he makes continual references to our being in the "first ... generation of software development." We've been doing software for almost 45 years, which puts us firmly at the end of the second generation, if not into the third. And that's precisely my point. Cleanroom attacks the "first generation" software development process while we're into the third. I know of many projects on which both a programmer and one of their parents worked on the same software. I have unconfirmed reports of instances in which a programmer and one of their grandparents worked on the same software.

versus that. The ultimate (so far) description of the software engineering process is Marciniak's monumental encyclopedia [MARC94].

I have distilled what I know about process into the following seventeen commandments [BEIZ95]. As you can see by comparing the two, the best practice shares many points with Cleanroom.

1. **Process roadmap:** everybody must know where they and their software are in the process at all times. The roadmap divides the process into steps that are easy to understand and control.
2. **Process control** means that there are mechanism by which the participants can learn how to improve those parts of the process with which they are most directly concerned.
3. **Quantification and metrics:** objectivity is prerequisite to process control. In engineering, quantification is the key objectivity. In software, metrics are the primary arm of quantification.
4. **Configuration control** means that at any instant of time, we can examine the product of our labor (a program, requirements, or test suite, say) and know: who, what, when, where, how.
5. **Requirements:** what are we building? Does everybody know? Does everybody understand the same thing? Here, I have to insist on documents because human memory is too fallible.
6. **Requirements traceability:** where did these requirements come from? Traceability means that requirements can be mapped onto software components and vice-versa.
7. **Strategic objectives:** for whom is this software being developed? What do they expect to get out of it beyond mere utility? There are hundreds of strategic objectives and management establishes the relative importance of each.
8. **Requirement validation criteria:** how will we know that a requirement is satisfied? What are the objective satisfaction criteria associated with every requirement?
9. **Responsibility:** who is responsible for what and when.
10. **Entry criteria**: a component can be passed to many different stages or be integrated with many other components, each of which may have different entry criteria.
11. **Exit criteria:** how do we know that a component is ready to go to the next stage of the development process? What are the tangible, objective, signs of completion? The exit criterion of a component is the union of the entry criteria for that component.
12. **Analysis:** analysis is the process by which a design evolves. Intuitive analysis, while often effective, cannot be easily communicated to others and consequently, formal, sometimes mathematical, analysis may be needed, even if only retroactively. This is where formal methods and proofs fit, if they apply.
13. **Design:** design should precede programming because it is less risky to do it that way. Design errors are paper errors and much cheaper than welded steel and concrete or congealed code.
14. **Design validation:** how do we know that a design will work if we haven't done something like this before? Validating the design is not the same as validating the implementation of the design. Validation is done by models, by prototypes, by inspection or by formal proofs where appropriate.
15. **Programming:** the act of coding a design and testing it to see to it that what we built is what we intended to build. The bricklayer checks courses with a level and programmers with a test.
16. **Integration:** only toy software has a system complexity equal to the sum of its component complexities. There is no integration worthy of the name without an integration plan and associated integration tests—be that the integration of low-level subroutines or quasi-autonomous systems.
17. **Testing:** we test because we recognize that as humans, everything we do, including testing, is prone to bugs; only execution of code, not models, are truly objective.

## 2. Process Comparisons

### 2.1. General

The following table compares Cleanroom and the best standard process. My objection to Cleanroom is rarely to what it advocates, but to what it leaves out. Many ingredients of Cleanroom did not originate with Cleanroom and are hardly unique to it [DECK94]: indeed, as section 2.2. below shows, they are in some form, part of the best standard process.

### 2.2. Cleanroom and the Best Standard Process Compared

The following table contrasts the key points of Cleanroom with the best standard process.

| | Factor | *Cleanroom* | *Best Standard Process* |
|---|---|---|---|
| | **Group 1—Essential Agreement** | | |
| 1. | Team Oriented | Yes | Yes |
| 2. | Incremental Development | Yes | Yes |
| 3. | Continual Integration | Yes | Yes |
| 4. | Stages, Hierarchical Build | Yes | Yes |
| 5. | Early User Involvement | Yes | Yes |
| 6. | Prototypes | Yes | Yes |
| 7. | Behavioral Specifications | Yes | Yes |
| 8. | Usage Specifications | Yes | Yes |
| 9. | Process Control and Feedback | Yes | Yes |
| 10. | Structured Constructs | Yes | Yes |
| | **Group 2—Equivalent Activity** | | |
| 11. | Design Validation | Yes | Design Inspections |
| 12. | Code Validation | Yes | Code Inspections |
| 13. | Increment Planning | Yes | Yes = Integration Plan |
| 14. | Pre-and post conditions | in "proofs" | As Assertions |
| | **Group 3—As Appropriate, Difference in Degree** | | |
| 15. | Formal Specifications | Yes | To Extent Possible |
| 16. | Correctness Validation | Yes | As Appropriate and Possible |
| 17. | Box Structure Model | Yes | One of Many Models, as Appropriate |
| 18. | Finite State Model | Yes | As Appropriate |
| 19. | Formal Validation (E.g., "Proof") | Attempted ** | As Appropriate |
| 20. | Stochastic Testing | Only Way | As Appropriate |
| 21. | Data Encapsulation | Yes | As Appropriate |
| 22. | Reliability Growth Models | Yes | Only If Theoretically Possible |
| 23. | Markov Chain Testing | Yes | As Appropriate |
| | **Group 4—Disagreement** | | |
| 24. | Programmer Compiles | No | Yes |
| 25. | Programmer Tests | No | Yes |
| 26. | Programmer Debugs | No | Yes |
| 27. | Test Validation | No ** | Test Inspections |
| 28. | Component Testing | No | Yes |
| 29. | Integration Testing | No | Yes |
| 30. | Partition Testing | No | Yes |
| 31. | Data Flow Model | No | Yes |
| 32. | Data Flow Testing | No | Yes |
| 33. | Stress Testing | No | Yes |
| 34. | Manual Syntax Checking | Always ** | Never |
| 35. | Cover Certification | Never | Always |

This table is divided into four groups:

1. Areas (items 1-10) in which there is no disagreement between the Cleanroom process and the best standard process. Since we agree, there's no point in discussing these points further except to note that none of these ideas originated with Cleanroom and none are unique to Cleanroom. I must point out, however, that with few exceptions [DECK94, PLES95], Cleanroom advocates like to promote the idea that all of the above (items 1-10) are unique to Cleanroom.

2. Areas (items 11-14) in which both the best standard process and Cleanroom agree as to objectives but differ in execution specifics. I will not discuss these areas either because the only point of contention would be which is more cost-effective. There is no substantive evidence one way or the other to indicate that, say, formal design validation is more cost-effective than formal design inspections. Every independent comparison such as Selby's study [SELB87] doesn't give us that level of detail and the comparison was never to the best standard process. What comparisons have been published have been total-process versus total-process. Since the wrong processes are compared (i.e., non-Orthodox Cleanroom against a bad "standard" practice), there's no point in discussing the specifics. The Cleanroom advocates assert the cost-effectiveness superiority of their method in these details with only anecdotal evidence for support: e.g., case studies. I'm content to let the ongoing evolution of the software development process decide the issue. We probably will have a shift to more formal methods in the future. I am not opposed to this, I welcome it. My objection is to the replacement of unit testing by such methods. Again, these components are not unique to Cleanroom [PLES95][2].

3. The third group (items 15-23) concerns methods used in both Cleanroom and the best standard process. The difference between them is that in Orthodox Cleanroom, these are the *only* methods used, while in the best standard process, these are *some* of the methods to be used if appropriate. As an example, the box-structured model is merely one many models that can be used in software development. The best standard process (as does Liberal Cleanroom) accepts all models and methods and lets cost-effectiveness in each specific instance dictate the extent to which it will and will not be used. By contrast, the Orthodox Cleanroom process is rigid, dictatorial, and insists on box-structured models. As before, nothing in this group is original or unique to Cleanroom.

4. The last group (items 24-35) are the points of specific disagreement and the only areas in which Cleanroom promotes methods that originated with it and that are unique to it. It is on these key differences that I will focus my criticism in the sequel. In this group, Cleanroom does *not* do something which *is* done in the best standard process. Our objection to Cleanroom has never been with what it advocates doing, but with what it advocates *not* doing.

The final point about the table is the asterisks that entries in the Cleanroom column, especially for item 19, Formal Proofs. The asterisks designate process steps or methods for which Cleanroom appears to assume a bug-free activity[3]. There are no asterisks following the best standard process entries because in the best standard process, all activities, every single thing done by a human, is assumed to have a finite bug probability. The bug-free assumption, about which I will have more to say below, is one of several fundamental weaknesses of the Cleanroom process.

### 2.3. Specific Points of Differences

I don't intend to discuss all the points of difference, because even the above table is incomplete on that. I will only discuss those points that I believe to be crucial and for which I believe Cleanroom has a potential for great harm.

1. **No Unit Testing**. Quoting Harlan Mills [MILL94]. "The increment must be of a size possible to develop without testing, say five thousand to twenty thousands lines of code...", " ...no unit testing by

2. "Cleanroom embodies a lot of processes that we didn't invent."

3. I say this because I have not been able to find a single instance in the openly published Cleanroom literature where the possibility of bugs in these parts of their process (e.g., formal proofs) is even mentioned, never mind discussed. Contrast this to the testing literature in which the bug possibility of test cases have often been discussed and explored both theoretically and empirically.

the developer..." and "The Cleanroom ... team does not test or even compile." Quoting Pleszkoch [PLES95]: "Development teams write the software without executing it.", "You can do better quality code without unit testing." and "Unit testing actually increases the [number of] bugs...more [unit testing] is worse." There can be no ambiguity over the fact that the Orthodox Cleanroom process advocates no unit tests by the programmers whatsoever. Along with that comes the fact that the programmer does not compile or debug their own code. That's my items 24-30 in the table. Since programmers don't test their own code, they have no test validation to do. The Cleanroom position on unit testing is explored in greater depth in section 3, below.

2. **No Integration Testing.** Cleanroom does not do integration testing. "Integration testing" is often confused with testing an already integrated system. This confusion is evident in the Cleanroom literature. Integration testing is defined as the specific testing done to explore incompatibilities between otherwise correct components[BEIZ90]. That is, integration testing explores the interfaces between components. It is only after integration testing has been completed that the components can be said to be integrated. While Cleanroom does advocate retesting components after integration (see Component Retesting below), there is no integration testing in the sense of the correct usage of the term. The Cleanroom's position on this is that their use of incremental builds and retesting the larger build obviates the need for integration testing. This position makes it clear that have no idea of what "integration testing" means, and like so many people who are ignorant of testing, they confuse it with testing an already integrated aggregate.

3. **Component (Re)Testing.** A component is what we call a unit after it has been integration tested and therefore integrated with its called components, subroutines, and functions. The best standard practice requires that the tests that were run under unit testing be rerun after the unit has been integrated to create a component. This is to assure that bugs stay fixed as we progress through the integration. Cleanroom *does* advocates retesting at every stage of incremental building, but there are serious questions (see below) as to whether or not such retesting is effective.

4. **No Partition Testing of any Kind**. One of the most dangerous characteristics of the Cleanroom process is its total reliance on stochastic testing at the expense of partition testing. Partition testing is the kind of testing we do when we check every feature and do enough testing to assure that every part of the code has been exercised. One broad division in testing is stochastic versus partition testing. Cleanroom prohibits partition testing. The best standard process uses *both*, as appropriate and effective. Partition testing does not imply testing all possible inputs (a straw man offered up in the Cleanroom literature [PLES95]) nor does it imply testing all possible paths (another favorite straw man of the Cleanroom crowd): both of these are stupid. Partition testing does mean a case-by-case set of designed and executed tests intended to probe every significant aspect of the software in absolute terms rather than in statistical terms. I won't go into further details here on partition testing because it has taken me and others several books to define that: BEIZ84, BEIZ90, BEIZ95, MARC94, MYER79.

5. **No Cover Certification** .
   Cleanroom advocates do not believe in the use of coverage tools. When the term "coverage" appears in the Cleanroom literature, it is invariably in a pejorative sense, as in "coverage testing." For more on this Orthodox Cleanroom position and the difficulties with it, see Section 4.6. below.

## 3. The Cleanroom Notions of Testing

### 3.1. A Common Thread

If one common thread runs through the Cleanroom literature, it is in the almost universal condemnation of "unit testing." Why the quotes? Because this term, as used by Cleanroom advocates means something totally different than the way the term is defined and used in the literature. More than anything else, this condemnation of unit testing is the most dangerous part of Cleanroom and the part on which I will focus.

### 3.2. What is Unit Testing?

Let's make it clear by what I (and IEEE standards) mean by "unit testing." A unit is the smallest compilable, testable segment, usually the work of one programmer (as originated), and does not include

called subroutines and called functions. Typical unit size ranges from 50 to 1,200 lines of source code, with the industry mean at about 300. I have heard people claim that they do "unit testing" but their so-called units were agglomerations of hundreds of routines comprising over 100,000 lines of code: that kind of "unit testing" doesn't work. Mills [MILL94], in talking about increments of 5,000 to 20,000 lines of code was not talking about units as defined above. If the Cleanroom advocates' notion of unit testing is such big aggregates, then they too have never really done unit testing and it is no wonder that such "unit testing" did not work for them.

We do unit testing because: (1) it's a lot cheaper to get bugs out in unit testing than in system testing, even if system testing is fully automated, (2) people should be responsible for their own actions, (3) it is more efficient at finding unit bugs than independent system testing [WONG94]. The first point is illustrated by the fact that the three most notorious bugs of the past ten years, the DSC switch bug, the AT&T switch bug, and the Pentium bug were all bugs that would have been found by proper unit testing and in which extensive stochastic testing did not find the bug. As for responsibility, the Cleanroom process exempts the programmer from ultimate responsibility because the remaining bugs will be found in a later phase by some other person. That increases the cost to detect the bug and is in a profound sense, morally corrupt. And there's no point in saying that unit bugs don't exist under Cleanroom because they do and always will.

As another note on personal responsibility, many leading software developers including those that had independent test groups are quietly dismantling such groups and folding them back into development groups. This can be done without risk or sacrificing objectivity, and with no loss in quality if the quality culture pervades the organization. The result, when successfully done, is less labor, higher quality, and shorter development time. Efficiency is improved because two people don't have to learn the same thing. Much smaller, specialized independent test groups handle such things as configuration and platform specifics, network testing, localization to a foreign language and similar areas of testing that require specialized operational knowledge. The best of us have progressed beyond blind independent testing while Cleanroom still advocates it as the panacea.

### 3.3. Testing Is Debugging

Either the Cleanroom authors do not distinguish between testing and debugging or they seem to believe that unit testing advocates make no such distinction. The term "testing" is almost always used in conjunction with "debugging" as in "test_and_debugging" [HAUS94, LING94].

| | |
|---|---|
| Dyer [DYER92] " | "...to forego programmer testing (debugging) of code..." |
| Cobb [COBB90] | "...Unit verification—debugging—is best done...." |
| Trammel [TRAM92] | "...testing (i.e., there was no debugging)..." |

and most clearly:

| | |
|---|---|
| Deck [DECK94] | " Such 'debugging' (sometimes institutionalized as 'unit testing') ..." |

Can there be any doubt whatsoever that in the minds of the Cleanroom advocates, as expressed in their liturgy, that unit testing is debugging and that debugging is unit testing?

This inappropriate equation, usage, and muddled terminology denies two decades of testing progress and the almost universal cry by testing advocates, that testing is emphatically *not* debugging. Indeed, as I and others have said too many times to cite, there can be no good testing and no good software as long as the programmers do not distinguish between testing and debugging. Since the Cleanroom advocates seem to be hazy about the distinction or seem not to have noticed the past twenty year's usage and literature on testing, let me remind them of the difference—paraphrasing Myers [MYER78] who was one of the earliest to clarify the distinction:

> Testing is done to expose (the symptoms of) bugs. Testing effort and planning is the same whether or not there are bugs.

Debugging is the process by which the location of the bug is determined and the process by which the bug is repaired.

Unfortunately, there are still too many software developers who (as do the Cleanroom advocates) do not distinguish between testing and debugging. The consequence of this is that they never really do good testing. If the Cleanroom advocates condemn such muddled practices, then I heartily join them. But, when they take a broad brush called "unit testing" or "testing" or "coverage testing" and apply it to the entire range of testing practices, to all of testing theory, then I must be adamant in condemning them.

The usage of these terms has been established in the literature and defined in the IEEE glossary of Software Engineering terminology. It is consistently used in the leading books on the subjects. And these standard usages are not the same as those utilized by the Cleanroom proponents. If they use variant terminologies and meanings, their argument is not substantive—indeed, it is propaganda rather than science. Since the Cleanroom literature is minuscule compared to the testing literature, we suggest that they bring their terminology usage in line with the dominant usage and not muddy the waters by semantic obfuscations.

### 3.3. "Coverage Testing."

A favorite pejorative term in the Cleanroom literature is "coverage testing." I was hard put to find out what was meant by that term because in most Cleanroom papers it is used without definition. Speaking to many different Cleanroom advocates and reading their literature, I came up with the following, contradictory, alternative definitions for this all-important pejorative term as used in the Cleanroom literature:

1. Any kind of unit testing.
2. Partition testing.
3. Testing done to satisfy unspecified coverage criteria.
4. Any testing based on structural models.
5. Exhaustive testing—testing all possible inputs [PLES95].
6. Testing all possible paths [COBB90].

This view, reiterated in many Cleanroom papers, is most clearly enunciated by Cobb and Mills in COBB90 (emphasis mine). Although there is no consistency in the Cleanroom literature as to the meaning of "coverage testing," the following definition seems to be dominant, and has the special attribute of being penned by Cleanroom's high priest and prophet, Harlan Mills.

"*Myth: Software is best tested by designing tests that **cover every path through the program***.

This testing method, called ***coverage testing***, requires that the test developer be completely familiar with the software's internal design."

Is this ignorant, venal, or both? No competent testing advocate proposes testing all possible paths. It is generally theoretically impossible, it poses recursively unsolvable problems, and it is a pragmatically stupid strategy[4] —all of which has been repeatedly stated by responsible testing advocates over the years. The above definition is wrong in several respects. It implies that there are knowledgeable testing practitioners, advocates, expositors, and theorists who support that myth—I don't know any and I haven't seen such an outrageous prescription in the literature published by people who know testing—although such statements by ignoramuses do crop up now and then. The above definition equates the use of coverage tools with testing all paths. It equates structural testing methods with testing all paths.

Interestingly enough, one of the most common mistakes made by newcomers to formal testing is to confuse branch cover with testing all paths. I have seen neophytes make this error hundreds of times. So much so that I have made it a point to emphatically emphasize that branch coverage does not mean testing

4. There are exceptional circumstances for life-critical software and for kernels where this exhaustive strategy may be justified; but such situations are rare.

all possible paths. My experience in this is not unique. Almost every expositor of proper testing methods has run across the same misconception and pedagogical problem. We can forgive neophytes for such error, but not people who purport to publish scientifically sound articles in the refereed literature. And we should likewise chastise reviewers and editors who let such canards get into print.

### 3.4. A Limited Testing Objective

The Cleanroom literature is clear with respect to *their* purpose in testing [TRAMM92]:

"The purpose of Cleanroom testing is not to find errors, but to certify the reliability of the software."

This is an unequivocal statement. Although I picked the above as the clearest and most compact Cleanroom notion of their purpose in testing, equivalent positions are reiterated in almost every Cleanroom document [DECK94, DYER92, HAUS94, IBMC93, LING94, MILL94...]. This central tenet of the Cleanroom dogma leads to a two pronged attack on unit testing. On the one hand, it allows them to dismiss any other purpose of testing (e.g., to find bugs) as unworthy. On the other hand, it allows them to reject any test technique that does not support their limited testing objective. From this we get a rejection of all partition testing methods, leaving only various (approved) forms of stochastic testing. I have even heard fervent believers express the idea that because a test technique does not support reliability determination, it isn't even testing. Another variant of this idea is their expressed belief that partition testing is not objective because it does not support reliability determination [PLES95B].

Cleanroom advocates are quick to point out that they are not opposed to testing per se: they oppose testing for any purpose other than reliability determination. The corollary to this is that for them, only stochastic testing has objectivity and utility. Thusly, with a broad sweep of the hand, they dismiss twenty years of research, 3,000 technical papers and reports, two dozen books, five annual conferences, several hundred undergraduate and graduate courses, and tens of thousands of practitioners of modern (non-stochastic) testing methods. Thus they dismiss the mainstream of software development. As Pleszkoch [PLES95] says: "Selling it [Cleanroom] has been a difficult sell." Is it any wonder?

### 3.5 No Integration Testing

Cleanroom does not advocate integration testing—indeed, if they know what the term means. Instead, they rely completely on repeated stochastic testing at every level of the incremental building process they advocate in lieu of integration. They assume that incremental builds and the stochastic testing (and copious retesting) thereof will find the bugs that would have been found by proper integration testing. The issue here is again one of cost-effectiveness and minimal coverage requirements. In the case of integration and integration testing, it is interprocedural interfaces that must be examined. In parallel to the commonsense notion that not testing a piece of code guarantees that you will not find bugs in that code, not testing an interprocedural interface guarantees that you will not find such interface bugs. Continuing the analogy, call-tree cover (both static and dynamic) is to integration testing what branch cover is to unit testing.

I do not claim that we have the answers to integration testing such as the best strategies for integration and for finding component interface bugs. Indeed, when it comes to integration strategies, i.e., the methods by which we select what specific components will be included in an integration increment and how to find the best sequence of such successive builds, our methods are probably as heuristic as the methods used by Cleanroom to select the components that they will use in *their* incremental build approach. However, we have had almost ten years of research on this problem and many excellent algorithms, especially for re-integration of software under maintenance, have emerged [ROTH94].

The increasing importance of integration bugs over unit bugs has become evident over the past several years, especially as a result widespread use of reusable components, component libraries, and object technology. We're getting the units under control and are learning to build high-reliability, trustworthy units. Now we find a dramatic shift in bugs resulting from incompatibilities between units. You can't do a stochastic test to find integration bugs because the integration issue is unknown and the joint user profile is often unknowable. Only detailed, specific, point-by-point partition testing of the interface, such as domain/range compatibility testing [BEIZ90] can find such bugs. The Cleanroom process as defined does

not acknowledge integration, integration bugs, and integration testing and therefore will be falling further and further behind. But since they fervently insist that finding bugs is not a proper goal for testing, my argument here is immaterial to them.

### 3.6. Testing Mainstream Avoidance and Testing Literature Ignorance.

A careful examination of the citations in published Cleanroom literature is instructive: especially if you do a citation index, as I did. Hausler cites Linger, Linger cites Hevner, Hevner cites Linger and Hausler, and everybody cites Mills, who copiously cites himself. The citation graph is very dense and strongly-connected within two small circles of about a dozen writers each. The first is centered on the IBM Cleanroom Technology Center while the second strongly-connected nodule of the Cleanroom literature is to various authors presently or formerly associated with the University of Tennessee, whose Computer Science Department appears to have made a passionate commitment to Cleanroom. Citations to authors outside of these groups are almost completely limited to authors of "approved" topics such as stochastic testing, formal methods, software reliability theory and other papers that appear to support the Cleanroom dogma. Of course there are always citations to icons of the industry, such as Boehm, Booch, Brooks, Hoare, Parnas , etc.

But where are the citations to the literature of the kind of testing that they so deplore? Only one writer, Dyer, has *any citations whatsoever* to the mainstream testing literature, and these were mostly out of date for a book published in 1992. Only 11 mainstream testing research papers are cited in DYER92. With the exception of myself and Myers, mainstream testing authors are not cited in any other Cleanroom authors' papers. I would not fault them for avoiding citations of authors who do not support their point of view, or for not mentioning the mainstream testing literature if they were not so unremittingly hostile to the mainstream. However, if you are going to criticize something in the technical literature, then the norms of scientific publication demand that you cite the authors you are criticizing and that you make your criticism specific. There are no such citations (Dyer's meager list excepted) and no such specifics. The criticism of the testing mainstream is always unfocused and vague: "...total path testing..., ...exhaustive testing..., ...coverage testing..., ...traditional unit testing..., etc." Such vagueness is appropriate to propaganda and sales literature—it is unacceptable in scientific literature and I wonder at the otherwise excellent editors and reviewers of our technical journals who have permitted such slovenly citation practices (or rather, lack thereof) for Cleanroom papers and for Cleanroom papers alone.

What is the consequence of this lack of citations? It is ignorance. They don't cite the mainstream testing literature because they don't read it and they don't read it because it is contrary to their dogma. The Cleanroom literature displays a pitiful ignorance of the mainstream testing literature. Their notion of testing practices in its most clearly enunciated form [DYER92] never gets beyond Myers' classic little gem, *The Art of Software Testing* [MYER79]. The Cleanroom literature typically describes testing as far more primitive than that espoused in MYER79. Myers' book, is of course a classic, but it is generally recognized as being out of date. When citations are made to my book, it is has been to the superseded, obsolete, first edition published in 1982 and not to the update of 1990. Neither Bender, Budd, Clarke, DeMillo, Frankl, Gerhart, Goodenough, Gourlay, Hamlet, Harrold, Howden, Korel, Laski, Marick, Marre, Miller, Morell, Ntafos, Osterweil, Ostrand, Podgursky, Poston, Ramamoorthy, Rapps, Richardson, Sneed, Soffa, Tai, Voas, Weiser, Weiss, Weyuker, White, Woodward, Zeil, Zelkowitz, nor Zweben nor any other mainstream authors of the testing literature appear as citations in the Cleanroom literature. My personal conversations and correspondence with Cleanroom advocates as a class, convinces me that for the most part, they know almost nothing about the mainstream testing literature and the practices advocated in that literature that they so constantly and passionately condemn.

### 3.7. Where is the Cleanroom Criticism of Testing?

Cleanroom is unrelenting in its attack of unit testing, and for that matter, with the sole exception of stochastic testing, they attack almost all other aspects of the mainstream testing literature. The norms of scientific discourse (as contrasted to sales hype and propaganda) demands that if something is to be criticized, it is to be specific and to the point; paper-by-paper and theorem-by-theorem; not by vague hand waving, semantic bickering, and innuendo. Where is the Cleanroom community's refutation of Gourlay? Where is their

rebuttal of Weyuker?[5] Where is their published refutation of testing theory? Where are their empirical studies that counter the empirical studies in favor of unit testing? Where is their refutation of mutation analysis? If unit testing and testing theory were such flawed concepts, there should have been by now, at least one published article refuting or seriously criticizing the mainstream; not by hand waving, but by the use of the same mathematical tools used by the mainstream researchers to erect testing theory. If unit testing is as flawed as they claim, then by now, they should have been able to publish at least one mathematically sound refutation. After all, they have been at it almost as long as the testing mainstream has been at it[6].

They have yet to find the flaws in Gourlay's reasoning; they have yet to question Weyuker's axiomatics; they have yet to point out (with mathematical precision) the fallacies inherent in the use of mutation analysis as a standard objective tool for evaluating test techniques. They have even failed to cite the copious critiques of these foundations that have been produced and published from within the testing community. This is in itself amazing—that they would forego the opportunity to use testing research's self-criticisms in support of their dogma. Why have they failed to critique testing? Because they are woefully ignorant. They probably haven't read our literature; and if they have read it, they haven't understood it—certainly not to the point where they could create a publishable critique in accordance to scientific publication norms.

## 4.Fallacies and Foibles of Cleanroom

### 4.1. General

Cleanroom is built on a set of fallacies. Most of these fallacies are assumptions about the world and the bugs therein that belie current realities.

### 4.2. The Bug-Free Assumptions

Perhaps the biggest contrast between Cleanroom and the best standard process is the tacit Cleanroom assumption that certain activities (e.g., formal proofs of correctness) are bug-free. Contrast this with the best standard process that assumes that *nothing* is bug free, including testing.

Among the things that can have bugs I offer the following short, albeit incomplete, list [LAWL95]: the proof is buggy, requirements have bugs, compilers and other tools may have bugs, programmers may use the compiler incorrectly, software may be installed incorrectly, proofs may depend on bugs in the proof support software, implementation of a "proven" spec may be incorrect, the programming language may have bugs, called and co-components may have bugs, component specifications may be buggy, configuration incompatibilities may exist, unproven changes and maintenance can introduce bugs.

The fallibility of formal methods, including those advocated as part of the Cleanroom process is the fact that they are not infallible—they are as, if not more, bug-prone than any other human activity [HALL90].

Informed testing advocates, including myself, do not oppose formal methods. We don't oppose it because we know that some problems will probably never be solved by pure testing methods. Primary among these are feature interactions and configuration compatibility. Other problems that are similarly combinatorially intractable are unlikely to be "solved" by testing methods—but by a combination of testing and formal methods. And the testing literature has been rich in its exploration of such hybrid approaches. The difference between Cleanroom and the testing mainstream is that the former does not address the issue of bugs in the formal methods they espouse while testers assume bugs as part of all process steps.

When I published an earlier version of this essay [BEIZ95B], this point was copiously criticized. "We don't believe that." they said. We cannot read their minds—only their published literature. And that literature is eloquently silent on the possibility of bugs in their formal validation methods.

### 4.3. The Bug Assumptions

Every quality improvement measure must make implicit or explicit assumptions about bugs. In Cleanroom's case, these assumptions are mostly implicit. The main problem with these implicit bug assumptions is that

5. That alone would take more people than the sum total of published Cleanroom advocates.

6. We can mark the beginnings of software testing theory by the seminal paper by Goodenough and Gerhart in 1975. The earliest paper on Cleanroom by that name appeared in 1981. Dyer [DYER92] dates it to 1980. After 15 years, a five year head start isn't that significant.

they correspond to the bug profiles that we used to find in small software years ago and not to the mega-KLOC programs of today. These are not surprising in Cleanroom's case because so far, the total published experience with this method has been on small-scale software.

1. **Emphasis on Control Flow Bugs.** This is out of date and applies mostly to small software built by novices. At best, today, this is about 5% of the bug count. Today, requirement bugs dominate (35%) and Cleanroom has no advantage over the best standard process because it promotes essentially the same measures for preventing and finding requirement bugs.
2. **Emphasis on Unit Bugs.** The implication is that most bugs are unit bugs. That may still be true today, but such bugs are easily found and removed by proper unit testing. The more difficult bugs (e.g., integration and system bugs), although less frequent than unit bugs today, are far more costly.
3. **No Mention of Integration Bugs.** The assumption is that if the parts work, then miraculously, so will the integrated whole. The possibility of individual components all being correct individually but not collectively, is not addressed. Integration bugs, especially in the light of reusable component technology, is the fastest growing bug category.
4. **No Mention of Data Flow Bugs**. There's no mention of data flow testing, data flow issues, or data flow bugs. Also important for object oriented programming. Also, probably the third largest bug category in modern software.

### 4.4. A Very Strange Activity

The strangest thing advocated by the Orthodox Cleanroom process was its insistence on manual syntax checking. The justification for this was given as the very act of syntax checking exposes other kinds of bugs that would not otherwise be seen, or words to that effect. What hogwash! I programmed in that ugly time when manual syntax checking was necessary because the syntax checking done by compilers ranged from nonexistent to useless. As Norm Abrams said on *This Old House*: "They don't build them that way any more—Thank God." In fairness to the current crop of Cleanroom advocates, only a few of them still espouse this bizarre position.

They don't push this idea (much) these days, but then they're not quick to refute it either. In analogy to religious cults, the later practitioners eventually drift away from the rigid orthodoxy of the founding prophet. If this idea, which was once held sacrosanct by the Cleanroom orthodoxy, has been rejected as impractical, inefficient, and flawed, what other ideas of this religiosity should likewise be questioned?

### 4.5. Statistical Problems

#### 4.5.1. The Issues

Cleanroom depends on valid user statistics, known as "user profiles." Given the above data, can we use it in a statistical model from which we can make statistically valid inferences about failure rates? Without sound statistics, the infrastructure of the Cleanroom process collapses. It collapses because the resulting failure rates determined by stochastic testing may have no relation to reality. If there is no corresponding relation to reality, then the risk becomes unacceptable.

I'll put the statistical knowledge questions another way: (1) Do we know how the users will actually use this software and (2) given that, can we make statistically valid inferences about the software's predicted failure rates and therefore know that it is fit to be used without further testing? The answer to the first question is "Hardly ever" and to the second, "It all depends," as I shall show below.

#### 4.5.2. User Profile

The most successful applications of stochastic testing to date have been in the telephone industry; the industry in which many of the more popular software reliability models and theories originated [MUSA90]. The success of stochastic testing in this industry often blinds its practitioners to how special that industry is, especially with regard to obtaining valid user profiles. We know more about telephone usage statistics, in the broad and in detail, than probably any other human activity. Furthermore, we have been gathering such data from the time the first automatic telephone exchange went live for the Danish PTT in 1910. User

profile data is easy to obtain for telephony because it is obtained in part as a byproduct of the system's operation, in part from billing data, and because such data is needed to tune these systems—and these are very highly tuned systems. Good usage statistics are also available for other applications, of which avionics control is the next best area. In avionics and other control systems, behavior is constrained by the laws of physics and by well-established operational flight profiles. Instrumentation for the important data is needed for tuning support. And there is an extensive flight testing program from which such data are obtained.

Does it follow from the success of statistical models and the ready availability of suitable statistics in the above instances that such data can be obtained for software in general? And if the underlying statistics is not available for an application, does it not follow that the statistical infrastructure of Cleanroom is suspect if not dangerous for that application? Here is a sample of the problems people have in getting user profile statistics in other than telecommunications, avionics, or control applications.

1. **No Prior Instrumentation**. The application is new or the package is new and there was no suitable instrumentation in the previous versions (assuming that there was a previous version, that is). We can't get suitable data until this software is operational. Therefore, there can be no valid statistical prediction of failure rates and therefore, there can be no trust in stochastic testing as the sole form of testing.

2. **Unstable Profile**. It is a fact that unlike telecommunications and control systems in which software and feature changes have little or no immediate impact on user behavior, in most applications, the features added or modified in new releases can engender big and often unpredictable changes in users' behavior. Early trials with beta releases don't help because beta testers are notoriously atypical. The same can be said for internal users (alpha testers) or for usage data obtained from most prototypes. When we add a grammar analyzer to a word processor or a data base function to a spreadsheet, say, we have no idea of how users will really exploit it and how the presence of a new feature will either increase or decrease the utilization of preexisting features. If you're dealing with software that will be installed on hundreds of thousands of platforms, you have a chance to find out. If not, say you install only one site, a few dozen, or a hundred, then you are unlikely to know your future user profile. Therefore, you have no idea of what risk you are incurring if you rely solely on stochastic testing.

3. **Untuned System, Too Late.** Another characteristic of most systems is that the usage profile changes in the first few months of use. Users play around with the new features or feature changes before they settle down to the new pattern. It is for this reason that software and systems should never be tuned until they have been in operation for several months. If we can't tune the system until it has been running for several months, how can we have valid user statistics on which to base our stochastic testing? Whatever valid data we get is too late to affect the development process.

4. **Too Complicated**. It's one thing to instrument a telephone exchange, or a control system, but quite another to instrument the typical application. Speaking from experience (most of my experience is in telecommunications), communications and control are far simpler from the point of view of profile data than, say, a word processor or a data base package. It is not enough to merely record the frequency of specific actions such as specific call types or command types: one must record and compile statistics on interactive scenarios and then use those recordings to create a non-deterministic finite-state model or more complex models based on stochastic regular expressions. Such models don't take samples of hundreds of actions, but of millions. Not only are the needed scenarios complicated in themselves, but there are many of them, and each one takes thousands or millions of runs to get suitable statistics. As an example, it took INTEL ***over one trillion tests*** to validate the model they used to predict the frequency of the Pentium bug [SHAR94]—and that scenario was much simpler than the typical user profile.

5. **Too Expensive**. If the instrumentation does not exist, it must be installed. There's an expense associated with that. Also, testing the instrumentation itself (I don't see how anyone could possibly test it stochastically) is harder and requires more thorough testing than normal software because so much is

riding on it [7]. If data are to be obtained from the field, then there is the possible question (and associated expense) of the operational disruption on the users of an instrumented system. The final part of instrumentation expense is the fact that operational profile data is often strategically important, may be classified, and must otherwise be restricted for security reasons. Security considerations alone can drive the cost of getting a valid user profile beyond all potential value. I have worked on projects in which obtaining statistically valid user profile data would have easily outstripped the cost of unit testing and stochastic testing based on that profile combined. I have not seen the cost of obtaining statistically meaningful user profile data addressed in the Cleanroom literature: there's vague mention of "an up-front investment", but no discussion of how big an investment. Is it that the Cleanroom folks are much better at getting profiles than I have been, much luckier, or is that they don't use statistically valid profiles even though they think that they do?

#### 4.5.3. Suitability of Statistical Models

The final statistical question concerns the validity of the software reliability models used in the Cleanroom process, and for that matter, software reliability models in general. The most detailed discussion of software reliability models vis-a-vis Cleanroom are in DYER92. The models described there are for the most part an uncritical transposition of hardware reliability theory to the software domain—a practice which has been too thoroughly discredited in the software reliability literature to rehash again. Software reliability is nothing like hardware reliability for too many reasons to discuss without increasing this paper by another 10 pages. Software reliability theory, at its most complete form, is to be found in MUSA90. That theory is based almost completely on telephony experience, without consideration of how special (and beneficial to the analyst and theorist) that experience is. I will not repeat the extensive critiques of software reliability theory as it is. The reader is advised to read the important paper by Littlewood [LITT78] for a critique. Littlewood cites many problems with the current theories (there are many different software reliability theories), most of which have not been satisfactorily answered.

Software reliability theory has been successfully applied to large scale, loosely coupled software systems for which meaningful user profiles are available; e.g., telephony. Application to small-scale software has resulted in misleading results ranging from wildly optimistic to wildly pessimistic. Interestingly, the word "scale" does not appear in Musa's book. Here are some other problems with the current theory.

1. **Not Composable.** Unlike hardware reliability theory where you can use validated models to get the reliability of a board, say, from the reliability of its components, or a computer from its boards, drives, etc., there is no statistically valid method for going from reliability figures obtained for subsystems to the reliability of a software system. While Dyer suggests just that approach in his book, it is not valid and is not based on any scientifically correct analysis. As it stands, the only way to obtain the reliability of a system is to measure the entire system and there is no known way to infer system reliability from component reliability.

2. **Bug Locality Assumption.** In hardware, the effect of a flaw or bug is generally local in time and space to the region in which the bug is. For example, a failure in an automobile transmission does not affect the radio or vice versa. This locality assumption is fundamental to all hardware reliability theories: it doesn't hold for software. Bugs in software have symptoms that are arbitrarily far removed in time and space from their causes. No published theory to date satisfactorily deals with this problem.

3. **Coupling.** Failures in hardware are assumed to be uncoupled, although here and there, some brave souls have attempted to analyze cascading failures in hardware. In software, two components can be perfectly reliable by any known measure, but yet incompatible, so that when they communicate with one another the system fails. Furthermore, in software, a component may not itself fail by any known test, but yet cause other components to fail. No known way to handle these either.

---

7. I have several times been called as a consultant on a supposed serious performance degradation of a communications systems only to find that the bugs were in the instrumentation and not in the system. The user profile data might be buggy.

4. **Distribution Assumptions.** All reliability theories depend on assumptions about failure distributions. The hardware distribution assumptions (e.g., exponential, weybull, etc.) do not apply to software.

5. **Scale Effect.** There are software reliability scale effects. The bigger the software system, the more loosely coupled it is, the better the theory seems to work; where by "working" I mean the theory's ability to predict failure rates based on measurements during testing. There has been no theoretical construct that allows us to take scale into account. The fact that software reliability theory has emerged mostly from large-scale, loosely coupled software and that it is applied by the Cleanroom advocates to small-scale, tightly coupled, software should give us pause.

The would-be user of the Cleanroom process who would risk his system, his company, his reputation, or perhaps someone's life on a highly controversial set of theories would do well to carefully explore the literature and read all the criticisms of that theory and not just the papers and books written by its proponents. At the very least, the would-be Cleanroom users should carefully and completely verify that the assumptions of the reliability growth models proposed for use in conjunction with Cleanroom do indeed apply their situation. It does no good to predict that the software will be ready for use on Thursday, August 24 at 11:07:06.534 plus/minus three centuries.

#### 4.5.4. Is Stochastic Testing Useless?

Is stochastic testing, as one of many test methods, useless in the light of the above criticisms and problems? Definitely not. If you want to make valid inferences about the probability of failure in use, then some kind of stochastic testing is essential. It is essential because you cannot, in general, make valid statistical inferences about failure rates from partition testing, no matter how thoroughly you do it [HAML88]. There is a rich literature on stochastic testing and how to do it right and for the right reason. Note that most writers about stochastic testing, myself included, are careful to point out that these methods should be used only *after* partition testing and never *instead of* partition testing as advocated by the Cleanroom proponents. This is a key difference between the best standard practice and the Cleanroom position: *after* versus *instead of.*

#### 4.5.5. The Retesting Problem

Because Cleanroom relies on stochastic testing driven by stochastic test generators, even if based on the same distribution, there is no assurance that the same test cases have been rerun. Unless *exactly* the same tests are run, using the same random number generator seed, there can be no assurance that regression testing of successive builds has actually taken place. Furthermore, even if the same seed is used, contemporary hardware and systems is operationally nondeterministic. Consequently, in realistic system testing situations there can be no assurance that the same tests were actually run during retesting. Since every source I read on Cleanroom did not address the issues of using the same seed in retesting or the nondeterminism of contemporary systems, it is safe to assume that these issues are not part of the Cleanroom process consideration. The same criticism can be levelled at rebuilding and retesting and after bug fixes.

The Cleanroom advocates counter this argument by saying that the sole purpose of testing is to obtain the software's reliability. A repetition of the stochastic test after the correction has taken place yields a new MTBF. When we realize how many tests it takes (even if fully automated) to achieve any kind of statistically valid measure of reliability, it means that we are effectively required to restart the entire stochastic testing process from scratch. The Pentium bug (see 6.5 below) took over one trillion test cases to validate the mathematical model of expected failure rates that they used [SHAR94]. This was a software, not a hardware problem. The software in question was equivalent to a few hundred lines of code. Proper validation of a bigger package (e.g., a few-hundred thousand lines of code) would take orders of magnitude more tests. And, in accordance to the Cleanroom prescription, such tests would have to be rerun for every incremental build and for every regression following bug fixes. It seems to me that a little judicious unit testing to verify the correctness of the fix would be in order here and more cost-effective.

#### 4.5.6. Target Reliabilities.

The target reliabilities in the Cleanroom literature are not impressive. Typically, numbers such as 0.995 are given as targets. Is this realistic? Hardly. On a typical day, using the SMARTDRV /S command, I can determine the number of 8kbyte disc accesses that I do: approximately 500,000 per day. It is not

unreasonable for me to demand that Microsoft provide a reliability for this (tiny, in comparison to the entire Windows environment) component such that I can be assured of less than one undetected failure per year. That works out to a reliability of 0.99999999333. If Microsoft wanted to assure all of its 86 million users that their software would not create an undetectable disc access error in 10 years of use, we will have to add about nine more nines. Practical considerations like this, market-driven considerations like this, show us that it is impossible to test software of any size (by stochastic or other means) to the reliability levels that rationality dictates. The Cleanroom advocates seem content to foist totally inadequate levels of reliability on the public.

#### 4.5.7. False Confidence

The bottom line of the above criticism of Cleanroom from a statistical point of view is that Cleanroom leads to false confidence. The possibility of statistically worthless analyses is too real for comfort. It is the worst kind of statistical legerdemain because it lulls its believers into thinking that they have tested well when in fact they have done no such thing. I have not, in any paper or book on Cleanroom seen any discussion whatsoever concerning the validation of the statistical models used, analyses of confidence intervals, T-tests, or similar standard statistical methodologies used to determine if a set of statistics warrants a statistically valid inference. The Pentium bug debacle (see below) shows how analysts can swing reliability results by five orders of magnitude based on which reasonable-sounding assumptions they make: that alone should give us pause concerning the statistical validity of the Cleanroom testing and reliability methods. Since the only kind of testing done under Cleanroom is stochastic, it may well be that superficial appearances to the contrary, software produced under this methodology goes into the field with no meaningful testing whatsoever, but with full (albeit unjustified) human, as distinct from statistical, confidence that it has been well-tested.

### 4.6. No Objective Measure and Coverage Tools.

#### 4.6.1. Why Unit Test Cover?

Many testing neophytes, and unfortunately, many programming practitioners, do not understand the purpose of coverage tools. The same misunderstandings and misconceptions abound in the Cleanroom literature. Using branch cover as an example, testing advocates have repeatedly stated that achieving 100% *branch* cover *in unit testing* is a *minimum mandatory requirement*. Actually, for typical C code, 100% predicate condition cover in unit testing (each predicate term of a compound predicate tested at least once TRUE and at least ONCE false) is a more realistic requirement. Let us look at the components of this sentence and see what is and what is not said:

1. **Branch (or predicate condition):** not all possible paths, not all possible inputs. The requirement that 100% branch (say) cover be achieved does not in any way imply or constrain the specific test techniques to be used to achieve it.
2. **In unit testing:** not in system testing, and probably not in higher level component testing. A rational criterion for statement cover (because branch cover is very difficult to measure in system testing) is about 85% during system testing.
3. **Minimum mandatory requirement:** this is where the biggest misconceptions abound. We have stated this as a "minimum mandatory requirement" and not as a goal to which the programmer should aspire. Units that are tested only to achieve branch cover, say, are probably ill-tested. This is a *minimum* requirement because doing less guarantees that bugs in the uncovered code will ***not*** be found. No knowledgeable, responsible, testing advocate today believes that merely achieving 100% branch cover in unit testing is good testing.

Good unit tests should be written to probe requirements without consideration of whether or not such tests achieve the desired cover level. If requirements are reasonably complete and well-documented (at every level—especially at the unit level—then a good set of tests against those requirements should achieve branch and predicate cover. The primary purpose of the coverage tool is to provide a metric that objectively indicates what was and what was not tested.

Three conditions have been formally shown necessary to find a bug [MORE90]: (1) the location(s) at which the bug exists must be executed, (2) the bug must cause an infection of a data state, (3) that infection must propagate to a meaningful output where it can be detected. Although satisfying these three conditions does not guarantee that a bug will be found, *not* satisfying any one of them guarantees that the bug will *not* be found. This is a formalization of the commonsense notion that if you don't try it, you can't find the bug. The purpose of using appropriate cover certifiers at every level of testing is to assure that the first of these conditions is always satisfied. Coverage metrics, as measured by an automated tool, are the primary metrics of test completeness.

It is instructive to note the experience people have had with coverage metrics and to distinguish between a programmer's subjective notion of coverage and reality. Most programmers when explained the kind of cover wanted (typically branch cover) and asked what cover they achieved in their unit testing, typically assert values between 95% and 98%. When the testing is repeated under a coverage tool, we find contrary to their subjective beliefs, that the best achieve only 85%, the typical is 60%, and the worst is 30%. We should ask what kind of coverage is obtained by the formal validation (i.e., "proof") methodology of Cleanroom. Why should we expect any better at this more difficult task than for testing? Why should we have a huge disparity between the programmers' subjective beliefs on the one hand (test coverage beliefs) and reality (test coverage measured by a tool) in contrast with formal proof coverage on the other? Because there is no tool to objectively measure proof coverage and because the process is less familiar to most programmers, we should expect an even greater variance between the programmers' subjective beliefs and objective reality. But without an automated coverage tool, we can never know.

**4.6.2. System Testing Cover.**
Today's crop of unit testing coverage tools are practically limited to code segments of under 50,000 lines. Above that, the execution time of the instrumented software becomes excessive and the instrumentation itself may introduce artifact that lead to misleading coverage information. The practical approach in system testing is to use statistical profiler tools that measure block cover. Many such tools are commercially available, in the public domain, and included in many operating systems. Their use is widespread and has been for over ten years. Typical target coverage under system testing is about 85%. Figures as high as 95% to 98% have been achieved, but there is no evidence that such high target cover values were cost-effective. We have an unconfirmed report of a joint study by IBM and Sony which is now underway to determine the optimum, effective, target for coverage in system testing. The use of system coverage tools such as statistical profilers has never been mentioned in the Cleanroom literature.

**4.6.3. Models Versus Reality.**
Models, no matter how detailed, no matter how well-conceived, are not reality. The entire Cleanroom validation approach, whether based on box-structures, formal proofs, an inspection variant, or whatever else they might want to throw into the pot, is based on a model of the software and not the software itself. ***THE MODEL IS NOT THE SOFTWARE!***

The only objective reality in software is tests conducted on the real software. No mental exercises, no matter how well intentioned and no matter how well executed—no such exercise is objective. We know how programmers fool themselves over what they actually covered. The unit coverage tool provides objective reality it in unit testing. The system coverage tool provides objective reality in system testing.

Without a coverage tool in system testing, we have no objective idea of whether or not the software under test was actually executed; certainly we have no idea of what part of that software was actually executed or not. Because we (in the testing community) assume that our test generators can be faulty, and that we are covering only what we hope to cover (if there were no bugs), the possibility of bugs demands that we use mechanical tools to confirm that what we thought we did is what we actually did. Without such tools, we are at best thinking that we are testing a model of a system that is in our heads and not an actual piece of code.

The Cleanroom advocates have not addressed this issue. They do not use coverage tools to objectify their testing. They do not take into account the wide disparity between what the programmers (and presumably the Cleanroom inspectors) believe they covered versus what they actually covered.

**4.6.4. A Modest Proposal.**

I offer the following modest proposal to the Cleanroom advocates. Use your stochastic testing methods to validate an important piece of software to whatever level you deem appropriate: say to a 0.999995 reliability. However, permit me to incorporate a suitable statistical profiler during your tests. You need not use this tool or the data it produces. When you have finished testing an incremental build to your satisfaction, I shall use the information gathered by the coverage tool and remove any code that was not covered in your tests. How can you, in the light of your professed dogma, possibly object? The code I propose to remove is code that you deem to be statistically insignificant. It will not, it cannot, according to your theories affect the failure rate you measured. It is statistically inexecutable (by your measure) and not important (by your dogma) and therefore, you cannot possibly object to my removing it. In fact, in order to be consistent with your philosophy, not only should you not object to my removal of this code, you should insist on it—in fact, consistency with your own principles dictates that you should have removed it yourself.

1. According to you, it need never be tested, because it does not get statistically executed and it does not statistically affect the determination of MTBF.
2. Removing this (untested code) can never introduce bugs, because code must be executed to manifest bugs—or do you believe that we should be concerned with bugs in code that is never executed—a kind of psychic, action-at-a-distance, bug?
3. Since there is a finite probability of bugs in this code and there is a probability (no matter how small) that this code could be executed, removing such code can only improve the software's MTBF, never decrease it. Why then, was not unexecuted code removal part of the Cleanroom process?
4. Having removed this unexecuted code, I will retest the software to the same MTBF by reusing your tests, but from a different seed. I should achieve 100% cover now, or at least, my coverage should increase. If I do not achieve 100% cover, then I will test again to the target MTBF and again remove any unexecuted code. I will continue in this iterative reduction until either I get 100% cover or until all the code is gone, whichever comes first. You cannot object to this procedure, except to say that I could be over-testing and wasting time.

The result is the smallest possible program that is statistically satisfactory (according to your theories) and that meets the sole objective of testing according to the Cleanroom dogma. Can you be consistent to your beliefs and object to this experiment?

We now go to the final phase of the experiment, step 5. This software is the control software for a spacecraft and you are an astronaut, sitting on top of five million pounds of hypergolic fuel controlled by the software tested as per the above.

5. Push the GO button. If you will not, you do not believe what you espouse. If you do, I shall be grateful for the (albeit, expensive) self-destruction of yet another Cleanroom advocate.

**4.6.5. Cleanroom and Coverage.**

The Cleanroom advocates do not believe in coverage tools. Because Cleanroom literature does not address coverage tools at all, it is safe to assume that they do not believe in such tools. Because no coverage tools of any kind are advocated and presumably not used, there is no way to know if the code has or has not been executed under test and therefore there is only a statistical probability that bugs have been found. [8]

8. We were told [PLES95B] that a putative coverage tool called "Certify", produced by Software Engineering Technologies, Inc. has been produced and it is in use by Cleanroom advocates. As near as we can tell, it is not a coverage tool—the only coverage it measures is coverage of the markov usage model the tool used to generate stochastic test cases. Covering such a model, is not of course, the same as covering the code under test. Measuring model coverage is certainly not the same as measuring code coverage. Extensive searches on the WWW and postings to all discussion groups concerned with software quality and testing have failed to yield anything substantive beyond the above about this tool.

## 5. Is it a Proven Process?

### 5.1. Has it Been Proven?

The above objections to Cleanroom would be justifiably meaningless and should be brushed aside if there were empirical evidence that showed that this was a more cost effective process model than what the Cleanroom proponents call "the traditional" model, or even what I call "the best standard process." We engineers should be pragmatists and if something demonstrably works, then use it and let the theorists catch up with us later. Has there been sufficient empirical evidence to claim that Cleanroom works in this sense? Ballyhoo, yes; evidence, no!

### 5.2. Dozens versus Thousands

The evidence supporting Cleanroom is anecdotal. The same 8 projects touted over and over again in their published literature. Although 24 projects are cited, only the following eight have been described in the refereed literature. Here they are:

| | | |
|---|---|---|
| Martin Marietta Documentation Project | 1820 Lines | 0.00 bugs |
| USAF HH60 | 30 KLOC | |
| NASA CFADS | 40KLC | |
| IBM COBOL RESTRUCTURING | 85 KLOC | 3.4 bugs/KLOC |
| IBM 3090E Tape Drive | 86KLOC | 1.2 bugs/KLOC |
| IBM AOEXPERT/MVS | 107KLOC | 2.6 bugs/KLOC |
| NASA Satellite Control | 170 KLOC | 4.2 bugs/KLOC |
| Ericsson Telecom | 350 KLOC | 1.0 bugs/KLOC |

The first of these, at 1820 lines of code, is clearly a toy. By today's standards, so are the next four. The last two rank as big toys in today's programming world that regularly measures applications in millions of lines of code. They're certainly not mega-projects. But even this data is not without blemish. I had a hard time pinning down the number of line of code for these various projects because they seemed to change (usually they grew) with each retelling. I used the largest figures published for each, but note that some of the original publications were half the size. Also, I have an unconfirmed, and unfortunately unpublishable report that the Ericsson project was not an orthodox Cleanroom project. They compiled their own code and use standard unit testing, I was told off the record.

It seems to me that after 15 years we should have many more examples than the same old tired report of the IBM COBOL restructuring project that appeared in the very first Cleanroom paper (it was only 52 k of code in its first incarnation). As pragmatic engineers who are working on projects that have a potential of risk to the public, is it wise to make radical changes in our process based on such scant, anecdotal, and suspicious data?

The best standard process has been used for thousands and tens of thousands of projects. It has been refined through continual evolution. It is not a break with the past, but a continually improving process. Unlike Cleanroom, it is not dictatorial and monolithic but flexible and eclectic.

### 5.3 Is Cleanroom Better?

The figures you see in the last collum of the above list is the number of bugs per KLOC obtained by Cleanroom, taken from the Cleanroom literature. Note, in some cases, these are not actual bug levels, but targets and it is not always clear which is which. These are not impressive; either as achievements or as targets. It is at best comparable to what we can get with well conducted code inspections prior to testing [GILB93, WELL93]. And then of course, there's the question of cheating (see below).

### 5.4. The Cheating Problem

Does any one actually do Cleanroom? Has any one actually done it? This is to be contrasted to the claims that they have done it. The best standard process does not have this problem because coverage tools are used to obtain an objective measure of the extent to which testing has or has not been accomplished. There can't be any lying about that, unless one fakes the results and gimmicks the tools.

The only independent studies of the effectiveness of Orthodox Cleanroom published in a refereed journal of which I know are the studies by Selby et. al. [SELB87] and Basili [BASI94] at the University of Maryland: independent in the sense that the authors are not associated either with the IBM Cleanroom Center or with Harlan Mills' company, Software Engineering Technology Inc. The Basili study [BASI94] suffers from exactly the same flaws as the Selby study and because my criticism is almost identical, I will illustrate it with the Selby study.

I have specifically excluded published case studies such as TRAM92, and the many other repetitions of the same 24 or so projects case studies that have been published over the years. I do not include them because they were not even attempts at controlled experiments. My criticisms of the Selby/Basili studies is of the best the Cleanroom advocates have to offer in justification. The other typical case studies are even more vulnerable to the following criticism.

1. The Selby study was an attempted controlled experiment in which some participants used "standard" testing methods and others used Cleanroom. There is no evidence that they received any kind of formal training in testing techniques. There is no reference to books or papers on testing (not even MYERS79) and the scant description of the test methods used implies that the standard testers in the control group did not know much about testing beyond that. There is no mention of the coverage method imposed or the use of coverage tools. In other words, what they were taught about testing (if anything) was obsolete, they didn't know much about what they had been taught, they had little time to internalize and master even those obsolete methods, and no adequate test completion criterion was used. Talk about stacked decks!

   Contrast the Selby study with a comparable study done by Crandall [CRAN89] at Brigham Young University. The software was previously tested and debugged real telecommunications software taken from IBM applications. The participants had received extensive (one semester) training in a course based on MYER79 and/or BEIZ82. They had practiced these techniques and internalized them. Testing was done to a branch cover standard. Although the participants were all students, they discovered more bugs than the professionals who used good, but not the best, available unit test methods. The deck was *not* stacked in this study. The study was later extended over a more than ten-year period with over 100 projects involving over 45 companies, with comparable results [CHOY88]

Now consider the participants in Selby's study (and other studies) who are presumed to have used Cleanroom.

2. The Cleanroom participants in the Selby study did not do manual syntax checking, they were also allowed to do ordinary testing for "special critical cases" [SELB87]. Could not the "special critical cases" have incorporated all the cases that would normally be considered in unit testing? What controls were in place to assure that they actually used the Cleanroom process? What else about their practice was not Orthodox Cleanroom.

3. Most studies were done on the Washington Beltway or its operational extensions (e.g., IBM Federal Systems Division, CSC, NASA Goddard, DoD). The participants have good jobs in a region increasingly depressed due to downsizing of the military-industrial complex. They live in overpriced houses in a barren real estate market. If they have to change jobs, it means relocating and huge personal losses. They want to keep their jobs.

4. They are programmers. We can assume that they all have a PC at home, or effectively unlimited and uncontrolled access to the right kind of PC and/or work station. They all have, or can obtain, download, or steal a copy of the right compilers and will do so if necessary. Whatever the official process might be, and even if that official process theoretically prevents them from compiling and testing their software, there is nothing to stop clever determined programmers from circumventing the official process and surreptitiously carrying out their own, "incorrect" or "forbidden" process on the sly.

5. They have been told by their managers and/or professor (or they invariably learn) that they are part of an experiment. They assume that the manager or professor in question has a stake in making the

experiment a success. Verbal protestations to the contrary, they assume that if he is in charge, he believes in it and wants it (Cleanroom) to work. They will guarantee that it works, even if they have to use a standard process in addition to the legislated process to make it work.

6. They write, compile, automatically syntax check, test, and debug their code in the usual way, but at home or on the sly. Those who do not cheat quickly look bad compared to those who do and are subjected to peer, management, or professorial pressure over not being cooperative (although the programmers and management have diametrically opposed objectives for cooperations). Those that remain fall in line (as cheaters); the non-cheaters leave or are fired because they are either obstinate in their honest use of traditional methods or appear to be slow (compared to the cheaters) in mastering the new methods.

7. The result is a self-fulfilling prophecy. The very set-up of the situation guarantees that Cleanroom will be successful, no matter what the facts might be. Note that this is not peculiar to Cleanroom. Had the process experiment been to prove the superiority of "The Fried Linguini" method, the results would be comparable.[9]

The subjects were not naive, they were sophisticated and had a lot at stake (their jobs) in proving the outcome. There was no control. The experiments were not single, double, or triple blind[10]. There was no effective mechanism to detect if cheating did or did not take place. I do not say that the above scenario took place, merely that it could take place and that there was no effective mechanism (indeed, there cannot be) to prevent its occurrence.

Please don't attempt to refute the possibility of the above by saying that "Programmers are honest and wouldn't do that sort of thing." or other similar naive objections. Programmers are neither more nor less honest than the general population. The only thing we can say about them in this respect that would differ is that should they decide to be dishonest, they would be a lot more clever about it than a general group. In any social experiment, the possibility of cheating (whether deliberate or unconscious as in the notorious Hawthorne effect) must be considered as part of the experimental plan or else the entire experiment is close to worthless. No such considerations appear in the Cleanroom literature.

A cleanroom advocate (Pleszkoch) attempted to refute this possibility by pointing out one case in which the participants could not have compiled or unit tested their own code because it was written in a PDL for which there was no compiler. The counter-refutation to this is trivial. They could have written their routine in any convenient programming language such as Pascal. Compiled, tested and debugged (at home) by whatever means was convenient and then manually translated the correct code into the PDL, which was then presented for the "Cleanroom" treatment.

I don't imply dishonesty on the part of any investigator—merely naivete. I *do* suggest a kind of dishonesty on the part of the subjects, but everything considered, who can blame them. Given the same circumstances, I would do the same—as would most of you, if you are honest about it.

The issue is not whether the participants did or did not cheat. The issue is whether or not they could have. Their entire notion of a controlled experiment is flawed. Until they can answer to these objections, their so-called "controlled experiment" cannot be not taken seriously.

---

9. Contrast this to the standard experimental method for comparing test techniques, including, say, a partition testing method to a stochastic testing method. In legitimate testing research these experiments employ automated mutation analysis, which while not perfect, eliminate much, if not all, of the human element in the experiment. The Cleanroom advocates have never published anything regarding the use of this standard experimental tool to compare their testing method to unit testing; nor have they come up with an alternate, equally robust and sound approach to doing subject-free experiments; nor have they produced a criticism of mutation analysis. If the Cleanroom testing process was more cost effective, a properly conducted set of experiments based on mutation analysis should have been able to support that thesis.

10. A single-blind experiment is one in which the subjects do not know if they are receiving the medicine or a placebo. In a double-blind experiment, the person administering the drug does not know who receives what in addition to the single-blind condition. In a triple-blind experiment, the person who evaluates the outcome does not know who received what and by whom it was administered in addition to the single and double-blind conditions. It is a well-established principle in medicine and in the social sciences that any experiment that is not triple-blind is seriously flawed, if not worthless. The virtual impossibility of doing triple-blind experiments for software development practices means that other methods such as those of epidemiology (see below), must be used.

### 5.5. Impossibility of Controlled Experiments, Epidemiological Methods

Social experiments, as any anthropologist will tell you, are always vulnerable to the possibility that the participants do not do what they say they do. I do not see how it is possible to conduct a triple-blind experiment that compares different software development practices, never mind a double-blind or single-blind experiment. I would criticize a so-called controlled experiment that attempted to prove the superiority of proper unit testing over Cleanroom in exactly the same way. A controlled experiment is impossible. What experimental methods, then, are to be used?

We have an analogous situation in Epidemiology and Anthropology [BABB86, MAUS85]. You can't infect the water supply of half of a town with cholera and provide pure water to the other half to see if water supply has anything to do with the spread of this disease. You can't urge a high caloric diet on half of a Native American tribe and rigid sugar intake control on the other half to see if there is or there is not a genetic component to diabetes. Such controlled experiments may be possible in a penitentiary or a concentration camp, but not elsewhere. The only thing you can do is to measure everything in sight, make very careful observations of what people *actually* do (not what they say they do, because they often lie), make those observations by the most objective means you have, and then over a long period of time (decades) and for many, many, subjects (thousands and tens of thousands), you *may* (not *will*, note you *may*) be able to make a statistically valid inference about what causes what.

Note that an inference of causality is not the same as correlation. To say that A and B are correlated does not mean that A implies B or vice versa. A recent example shows how misleading and tricky this kind of statistics can be [PETE95]. The number of fir trees, the number of moose and the number of wolves are correlated because the moose eat the fir trees and the wolves eat the moose. It had been generally believed that the fir tree density determined the number of moose and therefore the number of wolves. Fifteen years of study shows that it's the other way around. The number of wolves control the number of moose who in turn control the number of fir trees.

Very tricky stuff, trying to prove causality. What have we got for Cleanroom? The number of cases is not the number of lines of code or the number of programs, but the number of projects: eight that have been published in refereed journals and 24 claimed over all. What causality inferences can be made from this data: absolutely none [MAUS85].

### 5.6. Working for The Wrong Reasons

Some of you might say: "Okay, there's a real statistical issue and Cleanroom hasn't been proven from that point of view, but there are still those projects (8 or 24 depending on how you count) that seemed to do much better—anecdotal evidence or not, that's still evidence in Cleanroom's favor."

When you say that something is better you have to ask "compared to what?" Was Cleanroom better than an awful process that did not include inspections, any form of systematic testing, coverage tools, or any reasonable analysis? Is it better than an outmoded Code-Debug-Design paradigm? Probably. Almost anything would be an improvement over a primitive process. The history of software engineering is full of examples of things that appeared to work for the wrong reasons; the first and most famous of these being Harlan Mills' Chief Programmer Team.

Is Cleanroom an improvement over the unfortunate process still used by 75% of software developers? Probably. But the best standard process, were it adopted by such organizations, would provide a far greater improvement at a far lower risk. If we went back to the table of section 2.2 and put in another column titled "Worst Outmoded Process" it would have a "No" for most, if not all, of the first 23 items. Cleanroom is obviously better than that, but that's not what the followers of the best standard practice do today.

## 6. Risk Assessment

### 6.1. What is the Appeal

I have often asked myself "What is Cleanroom's appeal?" How is it that after fifteen years of intense promotion, almost non-existent experience (and that mainly on toys), and very minor penetration into the mainstream of software development, never-mind into the best standard practices—despite its poor record

and manifest faults, why is it still around and why is it still so appealing, especially to management? You might well ask the same questions about cults and similar phenomena.

The appeal is in the promise. It promises to remove a task that many programmers dislike, systematic partition testing, and replace it with a fully automated stochastic testing method. Testing, properly done, at all levels combined, is known to consume 50% or more of the software development budget and schedule. If someone came along and offered you a method that seemed to reduce 50% of your labor content to 5%, wouldn't you listen? Wouldn't you find that appealing?

Cleanroom holds the promise of major labor reduction without having to learn too much that's new. Learning how to do formal inspections and how test properly, and more important, getting such to be part of the working programmer's intellectual tool kit, buying coverage and other tools and teaching programmers how to use them, as well as many other aspects of the best standard practice—all of these are difficult, expensive, and time consuming. Cleanroom, just as most religious cults do, promises near instantaneous salvation by adherence to a simple formula. If we want to understand Cleanroom's appeal to the technically naive or to desperate managers facing an unachievable schedule, then we had better investigate the similar appeal of religious cults.

### 6.2. Risk to the Public

As engineers, our primary concern should be risk to the public. Our secondary concern should be potential financial risk to the organizations that employ us. Risk to our profession and how it is perceived by the public must also be considered and personal risks, of course, cannot be ignored. Engineering is deliberately conservative in the light of the above risks. Engineering practices usually evolve, more slowly than most of us would like, but the process is evolutionary, with best practices eventually becoming the norm through diffusion. It is rarely revolutionary. Revolutionary process changes in engineering usually lead to death. I do not wish to fly in aircraft whose sole means of testing was an Orthodox Cleanroom process. I do not wish to become a statistic that resulted from the inadequate (and unmeasured) coverage of dubious stochastic testing. And what I do not wish for myself I do not wish for others.

### 6.3. Process Not Verified

The Orthodox Cleanroom process has not been verified by either controlled experiments or by the massive body of experience that is needed to demonstrate causality. There is no assurance that even in the instances of seeming success, that the prescribed, Orthodox, process was actually carried out. There is a minuscule body (24 projects, if that) of evidence that claims to support the dubious methodology, and most of that has not been published in refereed journals. What has been published in the technical literature, has been a rehash of the same 8 or so projects over and over again. Constant retelling of the same cases does not improve the evidence.

### 6.4. Process Not Verifiable

The process is not verifiable by so-called controlled experiments. The process will not be verifiable by proper statistical methods until and unless it achieves a body of adherents that is comparable to the best standard process. Here too, it will be impossible to verify "Orthodox Cleanroom" because it is unlikely that the adherents will use such a tightly constrained methodology. The best that can be proven over a period of decades and thousands of projects, is a correlation, and possibly a causality inference for those aspects of Cleanroom (if any) that make their way into general practice.

### 6.5. The Pentium Bug

The Pentium floating point arithmetic co-processor bug is probably the most notorious bug in the history of the computer industry and might (but probably won't) prove to be the most expensive. Based on what evidence there is openly available [SHAR94] it appears that this bug was a unit bug in the software used to download a table of constants to a silicon compiler. Proper unit testing of this routine to a predicate condition cover standard would probably have discovered this bug—I have asked for, but have not obtained either a confirmation or a refutation of this conjecture from INTEL. As it was, extensive stochastic testing of the routine, consisting of over 1 trillion test cases, was needed to confirm INTEL's estimate of the bug's impact, which was claimed to be once in 27,000 years for the average spreadsheet user. Conversely, IBM

doing a different analysis, came up with once in 24 days. The disparity in these analyses is five orders of magnitude. Furthermore, with five million Pentiums in operation, the bug was discovered by only two users. You can't say that the bug wasn't important to INTEL's stockholder because the corporation made a $475 million dollar write off to cover the possible expense of replacing those chips. INTEL's stockholders were not well served by testers who overly relied on stochastic testing of the complete system (the Pentium chip).

## 7. The Future of Cleanroom

Cleanroom has been around for more than fifteen years. It's diffusion, despite well-crafted publicity, has been minuscule. Many Cleanroom advocates admit that selling Cleanroom has been difficult [PLES95]. The reason for that, they say, is that Cleanroom represents a fundamental departure from the mainstream and that many of its tenets are counter-intuitive. I offer explanations for the mainstream's (thankful) rejection of Cleanroom: it has no proven merit; a naked emperor is very visible.

Contrast that with other parts of the best standard practice, such as formal code inspections, that over a comparable period of time have made massive inroads to the practice of software engineering. If Cleanroom has a future, it will have to earn it. Earn it in the marketplace. It will have to slug it out with other vendors of proprietary methodologies. Conversely, if it claims to be science, it will have to engage in scientific debate in accordance to the norms thereof: by formal proofs of the fallacies of mainstream testing research that they so copiously criticize.

Meanwhile, we should stop giving it free advertisement space in refereed journals and conferences, unless we agree to provide similar (albeit very scarce) refereed journal and conference space and publicity to comparable vendors of proprietary methods. It is time to stop giving Cleanroom the respect due to science until, if ever, such respect is earned.

## 8. References

BABB86 Babbie, Earl. *The Practice of Social Research* (4th Edition), Wadsworth Publication, Belmont California, 1986.

BAKE94 Baker, F. Terry, *Chief Programmer Team*, in MARC94, pp. 93-94.

BASI94 Basili, Victor and Green, Scott. *Software Process Evolution at the SEL.* IEEE Software, July 1994, pps. 58-66.

BEIZ82 Beizer, Boris. *Software Testing Techniques*, 1st Edition. Van Nostrand Reinhold, 1982.

BEIZ84 Beizer, Boris. *Software System Testing and Quality Assurance*, Van Nostrand Reinhold, 1984.

BEIZ90 Beizer, Boris. *Software Testing Techniques*, 2nd. Edition. Van Nostrand Reinhold, 1990.

BEIZ95 Beizer, Boris. *Black Box Testing*. John Wiley and Sons, 1995.

BEIZ95B Beizer, Boris. *The Cleanroom Process Model: Caution Advised.* Debate at the Society for Software Quality, Washington DC Conference; March 8, 1995, Tyson's Corner, VA. Also, Journal of The Society for Software Quality, March/April 1995. Entire Issue.

BLUM92 Blum, Bruce I. *Software Engineering: A Holistic View.* Oxford University Press, 1992.

BROO86 Brooks, Frederick P. Jr., *No Silver Bullet—Essence and Accidents of Software Engineering*, in H.J. Kubler, ed. Information Processing 86, North Holland, Amsterdam, 1986. Also in IEEE *Computer*, April 1987, pps. 10-19.

CHOY88 Choy, Siu Hung. *A Framework for Product Verification Testing.* MSc thesis, Computer Science Department, Brigham Young University, December 1988. Vern J. Crandall Advisor.

COBB90 Cobb, Richard H. and Mills, Harlan D. *Engineering Software Under Statistical Quality Control.* IEEE Software, November 1990, pps 44-54.

CRAN89 Crandall, Vern. *The Development of an Industry-Education Relationship in the Test Environment: The Novell-Brigham Young University Experience*. Sixth International Conference on Testing Computer Software, Washington, DC, May 22-24, 1989.

CURR86 Currit, P. Alan, Dyer, Michael, and Mills, Harlan D. *Certifying the Reliability of Software*. IEEE Transactions on Software Engineering, SE-12 # 1, January 1986; pps 3-11.

DECK94 Deck, Michael. *Cleanroom Software Engineering: Quality Improvement and Cost Reduction*. Address a Pacific Northwest Quality Conference, 1994.

DECK95 Deck, Michael, Private correspondence.

DYER92 Dyer, Michael. *The Cleanroom Approach to Quality Software Development*, John Wiley and Sons, New York, 1992.

FAGA76 Fagan, Michael E. *Design and Code Inspections to Reduce Errors in Program Development*. IBM Systems Journal Vol #, 1976. pps 182-211.

GILB93 Gilb, Tom and Graham, Dorothy. *Software Inspections*. Addison Wesley, 1992.

HALL90 Hall, Anthony. *Seven Myths of Formal Methods*. IEEE Software, September 1990, pps 11-19.

HAML88 Hamlet, Richard. *Partition Testing Does Not Inspire Confidence*. Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, July 19-21, 1988.

HAUS94 Hausler, P.A., Linger, R.C., and Trammell, C.J. *Adopting Cleanroom Software Engineering with a Phased Approach*. IBM Systems Journal, Vol. 33 #1, 1994. pps. 89-109.

HEVN92 Hevner, Alan R., Becker, Shirley A., Pedowitz, Lenard B. *Integrated CASE for Cleanroom Development*. IEEE Software, March 1992, pps. 69-92.

IBMC93 IBM Tutorial Lecture Notes. *Operational Profiles for Statistical Usage Testing*. Presented at the Fourth International Symposium on Software Reliability Engineering. IBM Corp., Gaithersburg MD, 11/3/93.

LAWL95 Law, Lawrence. Private correspondence.

LING88 Linger, Richard C. and Mills, Harlan D. *A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility*. Proceedings, COMPSAC 88 (12th International Computer and Software Applications Conference. Chicago, IL. October 5-7, 1988 pps 10-17. IEEE, New York.

LING94 Linger, Richard C., *Cleanroom Process Model*. IEEE Software, March 1994, pps. 50-58.

LING95 Linger, Richard C. *Cleanroom Software Engineering: Management Overview*. In STSC "Cleanroom Pamphlet", Software Technology Support Center, Hill AFB, UT 84056.

LITT78 Littlewood, Bev. *How to Measure Software Reliability and How Not To*. IEEE Transactions on Reliability, R-28, 1979.

MARC94 Marciniak, John. *The Encyclopedia of Software Engineering*, John Wiley and Sons, New York, 1994.

MAUS85 Mausner, Judith S., and Kramer, Shira. *Epidemiology: An Introductory Text* (2nd Edition), W.B. Saunders Company, Philadelphia, PA, 1985.

MILL94 Mills, Harlan D. *Cleanroom Testing*, in MARC94, pps. 97-103.

MORE90 Morell, Larry J. *A Theory of Fault Based Testing*. IEEE Transactions on Software Engineering, Vol. 16 #8, August 1990, pps. 844-857.

MUSA90 Musa, John D., Iannino, Anthony, and Okumoto, Kazuhira. *Software Reliability: Professional Edition*. McGraw Hill, 1990.

MYER79 Myers, Glenford J. *The Art of Software Testing*, John Wiley & Sons, 1979.

PETE95 Petersen, Rolf. Study reported in Discover, March 1995, page 20.

PLES95 Pleszkoch, Mark. Tape of debate at Society for Software Quality, Washington DC Conference. March 8, 1995, Tyson's Corner.

PLES95B Pleszkoch, Mark. Private comment.

# Workshop Test Infrastructures

**Allyn Polk**, SunSoft

## Abstract

*This paper describes a method for testing GUI applications via instrumented code and a function calling mechanism. This method allows development engineering the flexibility to respond to user feedback or to change the GUIs while providing a stable interface for testing the majority of the application. As a result, tests are robust and portable, having longevity beyond just the current product release. Limitations and future directions of this approach and "object-aware" GUI testing tools are also discussed.*

## About the speaker

*Allyn Polk is the Software Quality and Testing Project Lead for SunSoft's Developer Products award-winning C/C++ workshop Product. Prior to joining Sun in 1991, Mr. Polk worked at Xerox in Rochester, New York; he received his Master's Degree in Computer Science, Software Engineering, at the University of Massachusetts in Amherst.*

## Workshop Test Infrastructure (WTI)

Testing application functionality without using the GUI



## The Problem

- GUIs, often unstable through Alpha, result in high maintenance GUI tests
- GUI changes from release to release often obsolete GUI test suites
- Current generation ("object aware") GUI tools are not the solution
- Test suites need to be insulated from GUI changes





## The Solution

- Build test suites based on a stable interface to an application's functionality rather than its GUI
- Delay traditional GUI testing until the after the GUI stabilizes
- Separate testing the application's functionality from testing its GUI



## How it works

- The application is instrumented with test functions
- A special mechanism is used to invoke a binary's internal functions
- WTI tests drive the application using the mechanism to call test functions
- Synchronization achieved through socket communication (if needed)



# Automatic Generation of Novice User Test Scripts

David J. Kasik and Harry G. George
Boeing Commercial Airplane Group
P.O. Box 3707, Mail Stop 6H-WT
Seattle WA 98124
kasik@garden.ca.boeing.com

**ABSTRACT**

Graphical user interfaces (GUI's) make applications easier to learn and use. At the same time, they make application design, construction, and test more difficult because user-directed dialogs increase the number of potential execution paths. Therefore, testing has become a significant part of the entire development cycle. This paper considers a subset of GUI-based application testing: how a novice user exercises an application in an unpredictable manner. We discuss different solutions to the problem and a specific implementation that uses genetic algorithms to emulate novice behavior and automatically generate test scripts.

**Biographical Information**

David J. Kasik is a Senior Principal Scientist working as the Boeing Commercial Airplane Group architect for Information Systems software engineering, geometry and visualization, and user interface technology. He has worked in the commercial CAD/CAM systems development area and led the development of a sophisticated toolkit that included a user interface managment system, computational geometry engine, three-dimensional computer graphics package, and complex object data manager. His main professional interest is in improving computing systems processes by combining more basic technologies. He has an M.S. in Computer Science from the University of Colorado and a B.S. from the Johns Hopkins University.

Harry G. George is an Advanced Computing Technologist working in the Boeing Commercial Airplane Group Knowledge-Based Engineering organization. He has developed business systems, embedded systems, and assembler/compiler tools. He has managed efforts in enterprise modeling and software engineering methods. His professional interests include natural languages, programming language design, and analyst/programmer productivity. He has a B.S. in Biology, an M.L.S, and an M.B.A., all from the University of Washington. He is currently completing a B.S. in Electrical Engineering from Seattle Pacific University.

# Automatic Generation of Novice User Test Scripts

David J. Kasik and Harry G. George
Boeing Commercial Airplane Group
Seattle WA 98124

**ABSTRACT**
Graphical user interfaces (GUI's) make applications easier to learn and use. At the same time, they make application design, construction, and test more difficult because user-directed dialogs increase the number of potential execution paths. Therefore, testing has become a significant part of the entire development cycle. This paper considers a subset of GUI-based application testing: how a novice user exercises an application in an unpredictable manner. We discuss different solutions to the problem and a specific implementation that uses genetic algorithms to emulate novice behavior and automatically generate test scripts.

**KEYWORDS:** User-interface Software and Technology: *automated test generation, novice testing, dialog model specification, genetic algorithms.*

## INTRODUCTION

The art of user interface design and implementation has become increasingly important to the success of computer applications. Graphical user interfaces (GUI's) make complex applications visually attractive and accessible to a wide range of users.

Users can exercise an interactive application in a large number of different ways. In reactive, GUI-based applications, even more options are available because multiple widgets and paths can be active concurrently. This causes problems in the application test process, especially when the application is made available to a large user community. Unsophisticated and novice users often take paths that neither the designer, the developer, nor the tester anticipated.

Tools and techniques to improve application reliability and usability are becoming more widely available. Examples include GUI builders, user interface management systems, usability engineering, and automated test software.

As the tools have been deployed, the prime beneficiaries have been experts. An expert user or tester usually follows a predictable path through an application to accomplish a particular task. A developer knows where to probe to find the 'weak points' in an application.

As a result, application state transitions are designed and tested for predicted usage patterns. However, many applications become unstable when given to novice users because they follow unexpected paths and do things 'no one would ever do.' Such errors are hard to generate, reproduce, diagnose, and predict. Current methods (e.g. recruiting naive users, extensive beta testing) to find problems novice users encounter are manual and costly.

This paper presents a repeatable, automated technique for mimicking novice user behavior that can be applied to GUI-based applications. The technique works at run-time and is, therefore, independent of application design and development tools. Because we have automated the generation of novice user behavior, an application can be quickly driven through a large number of unpredictable paths early in the integration and system test process.

## MOTIVATION

Testing is the subject of an entire branch of computer science [2, 20]. As shown in Figure 1, the general process defines a complete project.

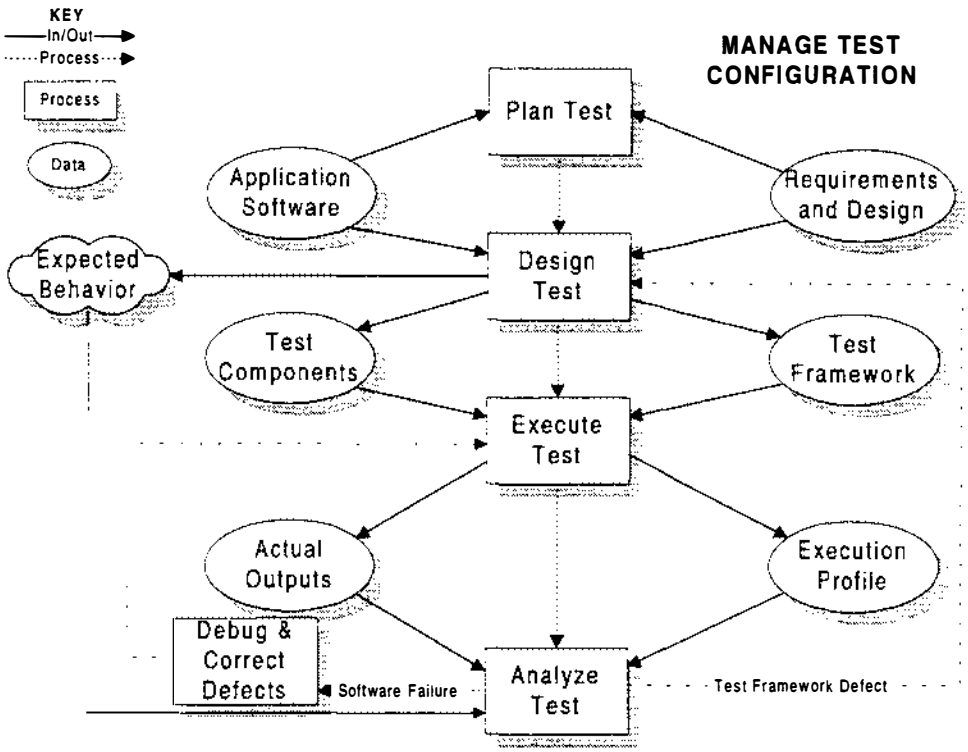



Figure 1. Test Process Framework

This paper deals with one aspect of the overall test process: how to automate generation of novice tests as part of the **Design Test** process. The importance of such testing can be observed by the size of beta test programs, such as Microsoft's 400,000 beta copies of Windows95.

GUI-based applications present special problems in test design. Older interactive applications often embedded command hierarchies directly in the structure of the program. The user then followed the command tree while navigating from one function to another. The test designer could assume that certain logical conditions held because the user could not deviate from the program-dictated sequence.

Multiple dialog sequences are available concurrently in a GUI-based application. The application becomes reactive and user navigation unpredictable. Therefore, a test designer must consciously assure that logical relationships are maintained despite:

- Interleaved function execution.
- Exit from a partially completed function.
- Asynchronous events generated by the user or program.
- Error conditions in the program.

The size and complexity of both the application program and the test set grow dramatically when concurrent dialog is involved. Industry statistics indicate that many GUI-based applications devote half of the code to managing the user interface [19]. As discussed in the next section, there are a number of different techniques for improving the user interface design/build process. Testing is still required even with improvements in the early stages of the development process. Mellor provides an excellent analysis of errors and their causes in complex, safety critical systems in [16].

The general problem of test automation for GUI-based applications has received some attention. The principal effort has been in the area of automated record/playback tools [25]. Such tools accept input at two levels:

1. Raw keystroke capture saves individual user keyboard and mouse events.
2. Logical widget names provide a higher level specification that is screen position insensitive.

The tools generally offer a language that can be used to program test scripts. Some form of bitmap compare is included to help a tester compare results of a test run with a known-good result.

Test automation tools work well when given to an expert and poorly when given to a novice because:

- An expert knows how an application should respond in a particular state, and a novice does not have such knowledge. In other words, an expert knows what an application can or cannot do as a task is being performed and a companion script written.
- A large set of test scripts is needed to faithfully represent how a novice might exercise an application. Generating those scripts by hand is tedious and needs to be re-done for each new version.

The rest of the section analyzes the problems associated with novice user testing.

**Novice Testing**

Testing can be conducted on a number of different levels to discover application faults and to measure and verify acceptable performance. The most common characterization of testing describes test phases where:

- *Unit test* attempts to find faults in a logical unit in isolation from others.
- *Functional test* attempts to find faults as logical units are connected together to perform specific functions.
- *System test* attempts to find faults as the whole application is used and to verify that the application meets requirements.

Different people are assigned responsibility for testing. Developers isolate specific application functions and interfaces during unit and functional test. These tests require extensive knowledge and analysis of code internals. System test is more task oriented. In addition to finding faults, system test determines if an application can be used to accomplish tasks. The testing is done in a black box manner and must be driven from the user interface. Experts perform most system testing, and beginners are occasionally recruited to find application weak spots.

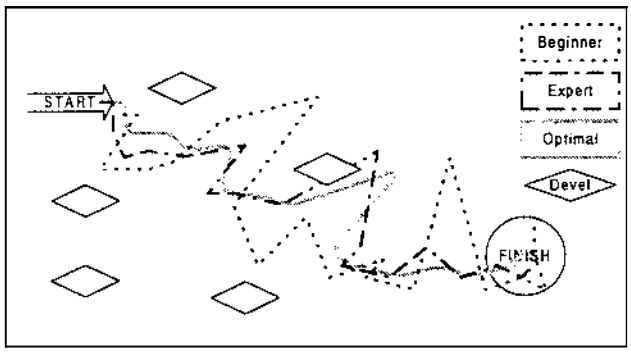



Figure 2. Different Paths through an Application

As shown in Figure 2, there are three paths that can be taken to perform a task during system test:

- The optimal path performs the task in the shortest time or fewest steps.
- The expert comes close to exercising an application in a semantically optimal manner. Experts know enough about program state to be able to work efficiently and define a reusable test script. Work has been done in the area of task analysis, cognitive simulation, and interface complexity with Goals, Operators, Methods (GOMS) models [15] to clearly define how an expert might use a complex application.
- The novice wanders in a reasonable but unpredictable manner. Novices gradually build a more accurate mental model of the application, which allows them to plan and traverse paths closer to the optimal. By the time novices can generate test scripts, they are often experts.

Novices do not work at random: they learn the application by performing a task. Random testing does have a role in exploring the outer limits of an application but is not our primary focus.

Novice testing is often ignored. At worst, some applications have been released in anticipation that users will find errors and accept fixes in the next version. Beta programs find some novice faults, but beta users are often quite literate in an application domain. Some companies recruit large numbers of naive users to test true novice behavior prior to beta release.

All three approaches to novice testing are costly. Not only do they occur late in the cycle, but a novice's actions are also hard to replicate. Novices wander through convoluted paths that only rigorous keystroke recording can capture. Large numbers of novice keystroke files become difficult to manage and upgrade to the next version.

We chose to automate generation of novice test scripts to address these problems.

## DIALOG SPECIFICATION ALTERNATIVES

Our approach assumes the existence of an automated record/playback tool. In order to generate scripts automatically for such a tool, we require a clear, accurate specification of both the user interface dialog and the state information that controls application response to user inputs. The implementation described in the next section depends on the dialog specification technique selected.

A user interface dialog specification consists of three components [23] whose relationship can vary because of changes in program state:

1. *Collector.* A collector groups logically connected functions. In general, collectors are used to assist a user in navigating through a set of functions or for finding a function in a larger set.
2. *Function.* A function identifies a block of dialog commands under control of input parameters.
3. *Parameter.* A parameter specifies a value or set of values that provide direct input to or control the execution of a function.

Three techniques can be used to specify user interface dialog and its state:

- GUI toolkits and application code.
- User interface management systems.
- Process modeling.

Each technique has advantages and disadvantages as the base for automated test script generation. Analysis of the alternatives led us to base our implementation on the most accurate and widely available dialog specification technique: a GUI toolkit and application source code.

### *GUI Toolkits and Source Code*

The most common form of dialog and state specification is the application program itself. The UI designer and implementer conventionally use a GUI toolkit to specify graphical layout and input gathering mechanics. The resulting applications are visually appealing and generally easy to use.

A GUI toolkit does not manage state information. Therefore, both logical relationships and concurrency must be managed in programmer maintained code. Application code grows more complex as the number of relationships increases. In addition, the way in which a programmer manages logical relationships and concurrency is different from but interleaved with the algorithmic code that performs a requested function.

Because the code is an integral part of the specification and state information cannot be derived from static code analysis, any automated test generation scheme needs to be able to obtain interface and program state information dynamically during execution.

### *User Interface Management Systems*

The second alternative is a user interface management system (UIMS). UIMSs have been formally pursued since the early 1980's [7, 14, 30]. The UIMS ideal is to operate the same application across a wide variety of interface styles (e.g., GUI, command line, form, script file) with little or no change to the dialog or application.

This is possible because a UIMS contains a dialog specification language that contains state information. The language is translated into an informal model that is interpreted to control program execution. Because they contain state information, UIMS specifications are inherently more complete than GUI widget hierarchies and more suitable for use in generating test scripts. The most effective UIMS-based approach occurs when the specification is used to generate both application state management code and test scripts.

However, few applications are specified with a UIMS. Applying any automated test generation technique to a manually developed application requires reverse engineering to derive the UIMS model. This is a daunting task: a working program often contains 'features' and loopholes that are difficult to capture in a more abstract UIMS model.

### *Process Modeling*

The third specification alternative for dialog is process modeling. There are a number of tools and techniques to formally model processes. Formal models are commonly used in time and state sensitive areas like:

- Computing hardware design [12].
- Real-time software systems [8, 32].
- Business process modeling [26].
- Hypertext browsing [29].

- Temporal logic for protocols [21].

In contrast to the informal models in UIMSs, formal models can be validated prior to execution. Validation methods include:

- Mathematical techniques. For formal methods such as temporal logic, proofs about the 'correctness' (e.g., internal consistency or consistency with specifications that are stated in a formal manner) of the specification can be automatically derived through theorem proving techniques [9, 18].
- System simulation. Finite state models and state-transition diagrams have been extended to hierarchical state diagrams to address the problems of concurrency [6]. The dominant underlying model is the Petri net [11, 27]. Petri nets are graphical in nature and the basic form can be analyzed mathematically or via discrete event simulation.

Formal models have been used as direct input to the test process, especially for safety critical applications. Two examples are path models built from source code to generate test data values [10] and finite state models to reduce the number of required tests [3]. Little has been done to automate the generation of the scripts themselves.

Petri nets have been used as a type of process model applied to dialog [1, 4, 22, 24, 31]. Palanque and Bastide have documented reasonably complex interfaces based on Petri nets. The nets proved to be an effective specification technique. However, they were manually verified and translated into an executable application.

Manual translation of a formal model to an executing program can contain errors and deviations from the specification. From a test generation perspective, the program contains the only accurate interface and state specification.

### Summary

Both user interface management systems and formal process models contain the information necessary to generate test scripts automatically from the specification itself. Neither technique is used widely. As a result, a reverse engineering tool would need to be built to produce either a UIMS or formal model specification. This approach would make the automated test generator itself difficult to use and error prone. Formal models also suffer from a lack of tools to translate the abstract specification into an executable form.

Therefore, we chose to work with the application itself as the dialog specification. Moreover, we chose to use the executing application rather than the source code. Source code analysis techniques cannot deduce the program state changes that occur during execution.

## PROTOTYPE IMPLEMENTATION

Given that an executing program specifies the application user interface dialog, we needed to develop methods to:

1. Allow the tester to invoke the novice script.
2. Simulate user inputs to emulate novice actions. Genetic algorithms, described later in this section, proved to be a good method to dynamically generate novice-like input events.
3. Capture the current state of the user interface during application execution. The implementation approach requires no added processing logic in the application source.

The prototype uses standard tools and techniques wherever possible. The prototype is application independent, and no specific application dialog design and implementation assumptions are needed.

### Tester's Interface

To simulate novice behavior, a tester:

- Begins with an expert generated test script.
- Inserts one or more **DEVIATE** commands into that script. **DEVIATE** departs from the existing script and then (perhaps) returns to the scripted path.
- Tunes the genetic algorithm parameters to build up a set of scripts that represent novice behavior.

The tester can use realistic, previously created scripts that contain novice behavior to initiate new simulations.

This interface strategy was chosen to give the tester the opportunity to control when deviations occur in a script. In this way, the tester can more thoroughly explore potentially sensitive application areas. An alternate approach, automatically invoking deviations in a random manner in an expert script, can be supported with the same implementation architecture.

Because the prototype works outside of an automated test tool, we defined a simplified script syntax for the expert test. A postprocessor generates scripts for commercial test tools. The resulting novice scripts can then be included as part of a complete test suite.

**Simulated User Inputs**

Commercial test tools provide a mechanism to drive a GUI-based application from captured keystrokes. Automation requires a method to generate keystrokes.

Analyzing user behavior led to the conclusion that emulating novice user behavior requires some method of 'remembering' success. Both novices and experts use an application to perform tasks. Novices explore to learn the semantics of individual functions and how to combine sequences of functions into meaningful work. Experts have already discovered successful paths through an application and rely on past experience to accomplish new tasks.

Random number generators alone are inadequate because they do not rely on past history to govern future choices. Genetic algorithms do rely on past history. Success has been reported in applying genetic algorithms to hardware test sequence generation [28]. Therefore, we chose to use genetic algorithms to simulate novice user events.

*Genetic Algorithms*

Genetic algorithms [17] can be programmed to simulate user input through a pseudo-natural selection process. There can be a variable number of genes in the pool. Each gene is composed of a fixed number of alleles that contain the 'genetic code.' A basic genetic algorithm:

- Initializes the alleles with valid random numbers.
- Repeats the following until its goal is reached:
  - Try and score all of the genes in the pool.
  - Reward the genes that produce the best results by replicating them and allowing them to live to a new generation.
  - Complete the pool in the new generation with genes containing random allele values.
  - Mutate some of the genes in the new generation.

Gene splicing, controlled mutation, mutation rates, and death rates can be programmed and varied. Genetic algorithms are not an effective way to explore all possible paths.

In a user interface oriented interpretation, the value in each allele value represents a different event that provides correct input to an active widget. This makes the input generation technique better than random because few keystrokes are wasted and cycles are eliminated.

**Run-Time User Interface Capture**

The techniques described in the previous sections are applicable to any GUI-based application. At run-time, different GUIs will require different implementations to capture the current state of the user interface as the Application Under Test executes. Our prototype, whose run-time architecture is shown in Figure 3, works with applications built with Motif 1.2 and X11R5. All software components were written as reusable objects in Modula-3 [5].

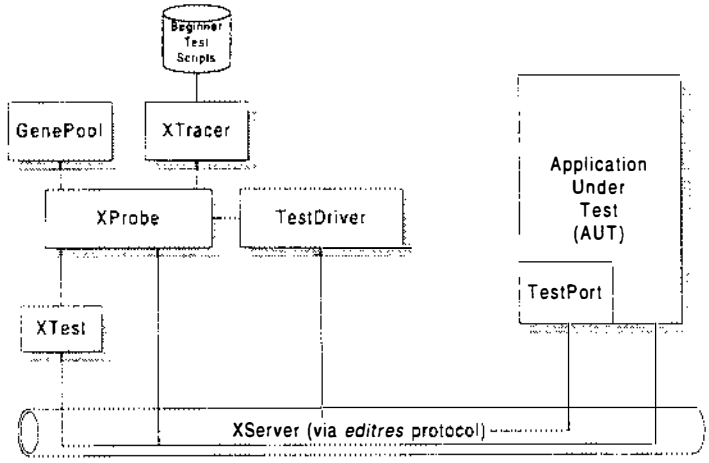



Figure 3. Run-time Architecture

The test script generator (XTest) can be applied to different Applications Under Test (AUT). During execution,

- TestPort and TestDriver establish the connection between AUT and Xtest.
- XProbe selects events and sends them to the AUT.
- GenePool uses the genetic algorithms to tune XProbe's selections and events.
- XTracer generates the genetically engineered test script at the end of a session.

XProbe, TestDriver, and TestPort use standard Motif and X communications with a protocol based on the *editres* protocol. This approach proved superior to other UNIX techniques for sharing information across process spaces like shared memory, rcp servers, pipes, and shared files.

The *editres* protocol asks the AUT to put some of its process-specific information into a selection for use by TestDriver. The prototype requires only one slight modification to the AUT source code. Two procedure calls must be inserted to establish the TestPort callback and to provide the ID of its top level window. These calls are executed only once during AUT initialization, and all other processing happens transparently. A preferable approach would require no source code change.

At run-time, TestDriver observes and controls the AUT as shown in Figure 4.

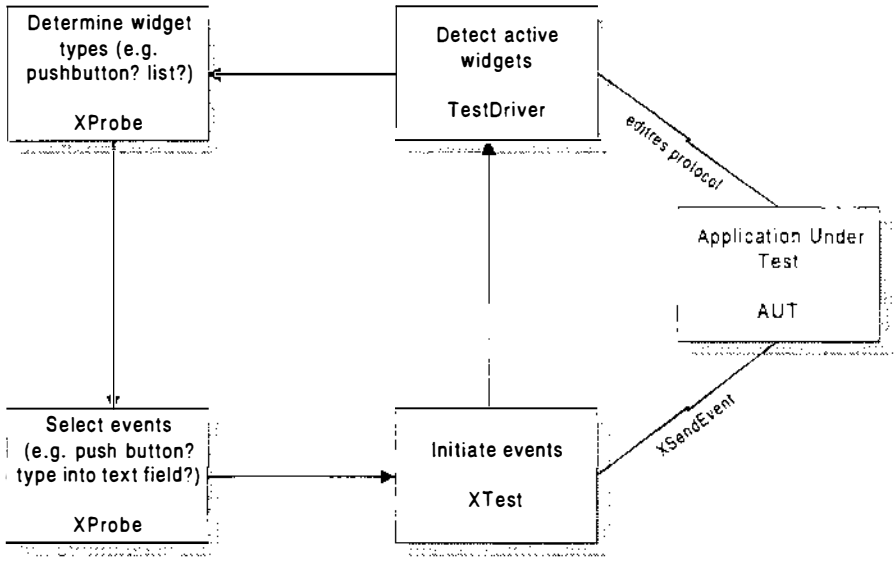



Figure 4. Observe and Control Loop

Observation is based on the widget tree active in the AUT. XTest requests that the AUT send its current widget tree after each event. The tree defines the set of legal actions at each program state.

Xtest controls the AUT through the standard X SendEvent mechanism. XSendEvent uses the window ID contained in the protocol stream. While this technique works, XSendEvent operates strictly at the keystroke level. The precise keystrokes needed to stimulate widgets and gadgets vary on an object-by-object basis. XTracer thus generates novice test scripts at the keystroke level. A reasonable extension is to implement a reverse protocol to pass widget-level information to the application under test and to generate a more readable test script based on widget names.

**RESULTS ANALYSIS**

The general implementation technique chosen proved to be flexible, adaptable, and application independent. The widget tree was correctly captured during execution, and keystrokes were successfully sent to different AUTs.

The biggest problem was development of a realistic reward system to let the genes simulate user events that seem like a novice. In other words, a technique is needed to let genes survive that 'behave' like a novice user. Because we used an executing application as the dialog specification, reward analysis was limited to the currently active widget tree. The widget tree contains syntax only, and rewards were created for entering data into a text field, choosing a menu or list item, selecting a radio button, etc.

The first approach we attempted let the genes try to learn how to become novices entirely through the reward system. Genes received higher scores when their alleles provided input to widgets on the same data entry panel or closed a

panel as opposed to widgets that caused a new menu item to be chosen. In this way, the reward system simulated a novice user who tries different parameter combinations before moving to another function.

At best, the scripts generated using the genetic algorithm alone exhibit chimpanzee-like, rather than novice user-like, behavior. Getting a script to actually accomplish anything meaningful was unlikely. This occurred because there is insufficient application semantic knowledge in the GUI widgets. A higher level specification is needed to insure that a particular run can even open a file.

The second approach produced much more interesting scripts. A tester can insert novice behavior at any location in an expert script with a **DEVIATE** command and proceed in either of the two manners shown in Figure 5. Pullback mode rewards genes for returning to the path, while meander mode allows the activity to wander in an unspecified direction.

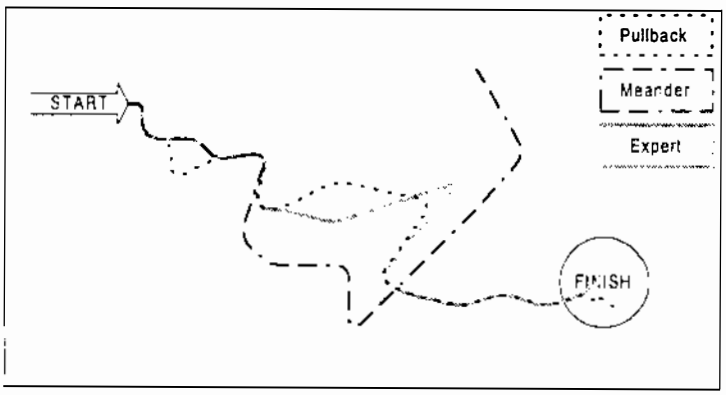



Figure 5. Deviation Modes

The reward system weights were changed to implement pullback mode. Pullback weights are based on the expert script's next action, which is defined by dialog shell and widget names. Higher scores are given to genes that select any widget in expert's next dialog shell. This emulates a user who remembers the right shell but not the right widget. Higher scores are also given to genes that select any widget with the same name as the right widget. This emulates a user who remembers the right name but not the right shell. Getting both the shell name and the widget name right pulls the user back to the expert script.

Our experiments demonstrate the ability to use a small number of inputs to generate a large number of test scripts in an hour or less. The resulting scripts have been informally evaluated to be representative of novice behavior by other staff members. We performed the evaluation both as the genetic algorithms were generating the scripts and during replay of the results.

Using automated script generation decreases the total number of scripts that need to be saved and modified as application versions change. Only the parameters that govern genetic algorithm execution need to be retained. Therefore, new novice scripts can be quickly regenerated for a new user interface and application.

### Remaining Issues

Our implementation resulted in a framework that can be used to emulate novice user behavior and solve some real problems. Additional work is needed to improve:

*Test script configuration management.* Our approach can be used to generate a large number of novice test scripts quickly. Measures must be developed to determine when enough novice testing has occurred.

*Test results evaluation and comparison.* The results of each novice test script must be evaluated to insure that the system has worked properly. At a minimum, the novice tests can be used to insure that the application does not break. More work is required to develop effective ways for determining that the results of a particular script produce the proper results. A simple comparison of the results produced by a companion expert test script is inadequate even in pullback mode. The deviation may have deleted or edited data and thus makes direct comparison impossible.

*Emulation and evaluation of more types of novice user behavior.* Additional genetic reward systems will expand the repertoire of available techniques. Care must be taken to more formally evaluate the results of any automated tech-

niques to insure their value and validity. The context for such comparisons is well documented (for a good example, see [13]). Usability testing facilities offer a good technology foundation for conducting real versus automated novice evaluations.

*Integration with automated test tools.* Given the current implementation of the novice test script generator, the use of genetic algorithms could be incorporated as a new command in any existing, widget-based test tool.

*Higher level user interface specifications.* In the long term, higher level dialog models than a GUI widget tree should be used to generate both application user interface code and test scripts. Automated UI code generation will decrease user interface state management errors (although application code errors will still occur). Automated novice test generation will be needed because more formal specifications do not contain enough information to define how a user really exercises an application to perform a task.

## CONCLUSION

Using automated test script generation to emulate novice users can inexpensively identify application faults earlier than beta test or production. Such faults are costly to fix and frustrating to users, especially novices.

Our technique works best as a companion to automated test tools and expert test scripts. Expert users still must generate complex scripts that exercise an application thoroughly. Genetic algorithms provide a controllable method of emulating novice input events to test an application in an unexpected, but not purely random, way. Including automated novice testing early in the development process should improve overall application quality.

### Acknowledgments

Rob Jasper and Dan Murphy of the Boeing Commercial Airplane Group have provided insight into the issues involved with testing and genetic algorithms and provided excellent comments on early drafts. Keith Butler of Boeing Computer Services helped in the review cycle.

## REFERENCES

1. Barron, J. "Dialogue and Process Design for Interactive Information Systems Using Taxis", Proceedings ACM SIGOA Conference on Office Information Systems (1982), pp. 12-20.
2. Beizer, B. *Software System Testing and Quality Assurance*, Van Nostrand, New York, 1984.
3. Bernhard, P. J. "A Reduced Test Suite for Protocol Conformance Testing", ACM Transactions on Software Engineering and Methodology, volume 3, number 3, July 1994, pp. 201-220.
4. Bordegoni, M., Cugini, U., Motta. M., and Rizzi, C. "An Environment for User Interface Development Based on the ATN and Petri Nets Notations", *User Interface Management and Design, Proceedings of the Workshop on User Interface Management Systems and Environments* (D. A. Duce, et. al., editors), pp. 231-245.
5. Harbison, S. P. *Modula-3*, Prentice-Hall, 1992.
6. Harel, D. "On Visual Formalisms", Communications of the ACM, volume 31 number 5, May 1988, pp. 514-530.
7. Hartson, H. R. and Hix, D. "Human Computer Interface Development: Concepts and Systems", ACM Computing Surveys, volume 21, Number 1, March 1989, pp. 5-92.
8. Hatley, D. and Pirbhai, I. *Strategies for Real-Time System Specification,* Dorset House Publishing, 1987.
9. Hoare, C. A. R. *Communicating Sequential Processes,* Prentice-Hall International, 1985.
10. Jasper, R., Brennan, M., Williamson, K., Currier, W., and Zimmerman, D. "Test Data Generation and Feasible Path Analysis", International Symposium on Software Testing and Analysis, Seattle WA, August 1994.
11. Jensen, K. "Coloured Petri Nets: A High Level Language for System Design and Analysis", Advances in Petri Nets 1990 (edited by G. Rozenberg), Lecture Notes in Computer Science, volume 483, Springer-Verlag, 1990, pp. 342-416.
12. JTC1-ISO/IEC Joint Technical Committee. *Information Technology-Programming Languages-CCITT High Language (CHILL)*, ISO Standard DIS 9496, 1994.

13. Karat, C-M., Campbell, R., and Fiegel, T. "Comparison of Empirical Testing and Walkthrough Methods in User Interface Evaluation", Proceeding of SIGCHI 92, May 1992, pp. 397-404.

14. Kasik, D. J. "A User Interface Management System", Computer Graphics (Proceedings of SIGGRAPH 82), July 1982, pp 99-106.

15. Kieras, D. E. "Towards a Practical GOMS Model Methodology for User Interface Design", Handbook of Human-Computer Interaction, M. Helander (editor), Elsevier Science, 1988, pp. 135-157.

16. Mellor, P. "CAD: Computer-Aided Disaster!", Proceedings of SOFSEM 94, October 1994, pp. 147-180.

17. Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 1992.

18. Milner, R, "Elements of Interaction", Communications of the ACM, volume 36 number 1, January 1993, pp. 79-89.

19. Myers, B. A. and Rosson, M. B. "Survey on User Interface Programming", Human Factors in Computing Systems: Proceedings of SIGCHI '92, 1992, pp. 195-202.

20. Myers, G. J. *The Art of Software Testing*, Van Nostrand, New York, 1978.

21. Ness, L. "L.0: A Truly Concurrent Executable Temporal Logic Language for Protocols", IEEE Transactions on Software Engineering, vol. 19 no. 4, April 1993, pp. 410-423.

22. Oberquelle, H. "Human-Machine Interaction and Role/Function/Action-Nets", Petri Nets: Applications and Relationships to Other Models of Concurrency Part II, pp. 171-190, Springer-Verlag, 1987.

23. Olsen, D. R. *User Interface Management Systems: Models and Algorithms,* Morgan Kaufmann, 1992.

24. Palanque, P. A., Bastide, R., Dourte, L., and Sibertin-Blanc, C. "Design of User-Driven Interfaces Using Petri Nets and Objects", Proceedings of Advanced Information Systems Engineering: 5th International Conference, CAiSE 93, June 1993; Springer-Verlag; pp. 569-585.

25. Parker, T. "Automated Software Testing", Unix Review, January 1995, pp. 49-56.

26. Peters, L. J. and Schultz, R. "The Application of Petri-Nets in Object-Oriented Enterprise Simulations", Hawaii International Conference on System Sciences (HICSS-26), January 1993.

27. Peterson, J. L. *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, 1981.

28. Rudnick, E. M, Patel, J. H., Greenstein, G. S., and Niewmann, T. M. "Sequential Circuit Generation in a Genetic Algorithm Framework", Proceedings of the 31st Design Automation Conference, June 1994, pp. 698-704.

29. Stotts, P. D., Furuta, R., and Ruiz, J. C. "Hyperdocuments as Automata: Trace-based Browsing Property Verification", University of North Carolina Research Report TR92-038, October 1992.

30. Thomas, J. J. and Hamlin, G. (editors). "Proceedings of the Graphical Input Interaction Techniques: Workshop Summary", Computer Graphics, volume 17 number 1, January 1983.

31. vanBiljon, W. R. "Extending Petri Nets for Specifying Man-machine Dialogues", International Journal Man-Machine Studies, volume 28, 1988, pp. 437-455.

32. Ward, P. and Mellor, S. *Structured Development for Real-Time Systems, volumes 1-3,* Yourdon Press--Prentice Hall, 1985 and 1986.

# The Quest for the Right GUI Test Tool

**Phil Allred**
pallred@novell.com

**Darren Scruggs**
darrens@novell.com

**Jim Gifford**
jgifford@novell.com

**Novell Applications Group**
Novell, Inc.
1555 N. Technology Way
Orem, UT 84057-2399
(801) 225-5000

## Abstract

When WordPerfect Corporation needed a GUI test tool to meet their needs, it evaluated several tools and decided to build its own. This paper reviews some of the requirements that are important to testing GUI-based application software in a cross-platform environment. These requirements include cost, context-sensitivity, synchronization, extensibility, client-drawn objects, and cross-platform compatibility.

## About the Authors

*Phil Allred* joined WordPerfect Corporation five years ago in software configuration management group. He managed Shared Code Testing department for two years and for the past year has been the Director of Engineering Services at the Novell Applications Group since October 1994. Prior to joining WordPerfect, Phil practiced law in Aberdeen, South Dakota. He has a B.S. in Finance and a *Juris Doctor* degree, both from Brigham Young University.

*Darren Scruggs* began working at WordPerfect in 1990 in the testing areas he wrote automated test scrips for both engine and interface level testing. In the fall of 1993, He began designing and developing AppTester and has been an Engineer on the AppTester development team since that time. Prior to joining WordPerfect, Darren spent two years developing applications targeted for the home market.

*Jim Gifford* is currently a software test engineer at the Novell Applications Group and is the lead engineer in testing automation for the GroupWare Division. Since he began his employment in 1989 he has served in a number of capacities including management. For a period of one and one-half years, he was Manager of Testing Automation at WordPerfect, the group which wrote AppTester. He enjoys testing automation because he considers this aspect of software engineering and quality assurance still largely a pioneering effort.

# The Quest for the Right GUI Test Tool

## 1. Introduction

In 1993, WordPerfect Corporation (now known as the Novell Applications Group and hereinafter referred to as "NAG") found itself on a sort of "quest" that many software development companies have found themselves on. NAG had a need for a GUI test tool to help in the automation of the testing of software such as WordPerfect, Presentations, and suites of application software. At times the issue seemed overwhelming. NAG did not understand all of the requirements of a good GUI test tool at first. Now that the company has a tool, Novell wishes to share some of those requirements with you.

### 1.1. What this Presentation is Not

This presentation is not to take on the debate about whether GUI testing is profitable, or even the extent to which manual testing can be effectively automated. The presentation is not on how to do a cost/benefit analysis. Nor is it the intent of the authors to downplay any existing tool. Rather, it is to give others the benefit of Novell's experience in forming and analyzing the requirements of a GUI test tool for use in testing shrink-wrap software. The authors hope the presentation opens others' eyes to some of the issues that NAG explored so that others can make an intelligent decisions when selecting a tool.

As you will see, NAG went through a make or buy analysis and decided to make its own tool, which is called AppTester. However, the purpose of the presentation is not to "sell" AppTester. In fact, the AppTester is not sold as a standalone product. As of this writing, there are plans to include it in the NetWare Software Developer's Kit (SDK), but NAG's Automated Testing department, which created the tool, is a cost center and will receive no revenue from the inclusion of the tool on the SDK.

It is also important to remember that the evaluation occurred in late 1993. The functionality of some of the tools has changed considerably since then.

## 2. Early Difficulties with Testing Tools

By 1993, NAG had used at least four different automated testing tools in various departments. Each tool brought its own set of frustrations, which can be classified into the following areas:

### 2.1. High Costs/Limited Availability

One of the company's early experiences with automation was on the UNIX platform. Because of the enormous expense of the test system that NAG was using, only five licenses were available for use among more than fifty testers. Sharing these tools was an experience somewhat akin to being a computer science student a few decades ago when one would stand in line to use the compiler on the main frame. The script had better work the first time because debugging was out of the

question. In fact, debugging had to be done by a different group than the group that wrote scripts.

The DOS testing tool which was used in the early 90's at NAG required a completely separate PC for the operation of the scripts. This made the tool too costly to supply all of the DOS testers with a license. The high costs made for limited availability.

### 2.2. Context-sensitive Nature of Tools

Many of the tools in use at the time were context-dependent. When changes occurred to the interface, however small, the scripts were rendered unusable. Testers could hardly keep up with maintaining the few scripts they wrote. This problem would also recur when it was time to test localized versions of our applications.

### 2.3. Synchronization Issues

Another important problem in NAG's early automated testing efforts was that of timing. If tests were recorded on a fast machine, they would get ahead of themselves when played back on a slow one. Static waits or pauses were inserted in scripts, but it was always a guess whether the wait would be the right amount for another machine. Inserting the waits also made the tests inefficient on fast machines.

### 2.4. Cross-platform Limitations

Of the four tools in use in the early 90's at NAG, none were cross-platform. Our revenues were generated by the dominant platform, but it was strategic to the company to provide applications that would work across many platforms. These other platforms had smaller revenues. It was recognized that the key to reduce the code of development on the platforms was to share code - not only development code but test code. Testers needed the ability to leverage their efforts on the dominant platform so that testing on other platforms was cost-effective. This was impossible with the existing tools at NAG. It was clear that NAG needed a single tool that would solve some of these early difficulties.

## 3. Initial Requirements

At the later end of 1993, testing managers at NAG elected to more aggressively pursue a comprehensive automated testing strategy. They wished to see automation play a bigger role in the process that testers used to assure quality in NAG's software products. To facilitate the forming of this strategy, a position was created called the Manager of Testing Automation and a Testing Automation Committee was formed to work out and document the details of this strategy. Paramount to the formation of an automation strategy was the selection of a GUI testing tool. The group of testing managers fashioned a list of requirements that the committee would use in the selection of a testing automation tool. These requirements were as follows:

### 3.1. Licensed Seat for All Testers

Due to past experience with a limited number of licenses among large groups of testers it was felt that in order for testers to be most productive, each must have unlimited access to testing automation tools. In other words, each tester should be able to write, execute and debug their own test scripts. It did not seem reasonable to expect testers to have to "schedule" the use of automation tools to perform the above mentioned tasks. This required the committee to find a technology that had an economical cost-per-seat.

### 3.2. Cross-platform Paradigm

NAG has traditionally offered its software product line on a variety of computer platforms and operating systems. In the last few years this scope has been narrowed to include Windows, Macintosh and UNIX. Since the majority of NAG products at the time of this study included graphical user interfaces (GUI's), which were very similar across these platforms, it was thought that the testing departments could benefit from a test tool built around a cross-platform paradigm. It was hoped that a test script written in one environment could be plug-and-played in another environment with little or no modification.

### 3.3. Testing Custom Objects

In addition to standard GUI objects, the testing tool needed to be extensible to allow the querying or manipulation of custom objects. These objects might either be custom derivatives of standard GUI objects or unique client-drawn objects where event mechanisms are built into the application. In short, the tool needed to have provisions for communicating with objects through hooks provided by the application programmer.

### 3.4. In place by January 1, 1995

Testing management required that the tool be ready to use on January 1, 1995. This gave the developers a little over a year to research and find a tool that would meet all of the above requirements. NAG had entertained from the beginning the possibility of developing its own tool if it did not feel like tools currently available on the market would meet our needs.

## 4. Investigation Stage

The first task that the Testing Automation Committee undertook was to survey a number of tools available on the market. The purpose of our research was to understand the capability of each of the tools and to then measure them against our list of requirements. This survey covered a period of about three months. Five tools were evaluated at length, most of which were already being used somewhere in the company. Other products were also considered for evaluation but were quickly eliminated because they did not have a cross-platform offering. Some single-platform tools were evaluated simply because of their prior use at NAG.

## 4.1. The tools that NAG evaluated

NAG evaluated the following tools:

- Microsoft's MS-Test 2.0
- Novell's NControl 2.6
- Segue's QA Partner 1.0
- Mercury Interactive's X-Runner and Win-Runner version 2.0
- Apple's Virtual User (VU)

It should be noted that since our evaluation most of these companies have produced later versions of their respective products and therefore our assessments of each product may not be reflective of current versions.

## 4.2. Matrix of Functionality

In addition to measuring each tool to see if it met our basic requirements, each tool was evaluated on the strength of its programming environment. Every test automation tool has two fundamental components in common: 1) a programming or scripting language that allows you to formulate logical test cases and 2) a set of test functions that allow a tester to query (get the state of) and manipulate (send an event to) GUI test objects. Each tool was evaluated in the following two areas:

### 4.2.1. Language Intrinsics

We evaluated each tool's language constructions. Did the tool provide a rich set of data and control structures? Did it provide most of the atomic data types (characters, numbers, etc)? Could user defined data types be created? How about decision loops and branching (If-Then-Else statements, FOR loops, WHILE loops, CASE statements)? Could advanced data structures be created using such things as pointers, structures, arrays, etc.?

### 4.2.2. Test Functionality

We evaluated each tool's GUI test application program interface (API). (See appendix 1 for a table comparing functionality of five tools.) How powerful where these tools in allowing testers to query and manipulate GUI test objects? Were the functions intuitive? Can the test API be extended? What about API's that are external to the test tool? What about test support structures such as log files, exception handlers and configuration files?

Additionally the integrated development environment was also considered. Did scripts have to be loaded into an interpreter or could scripts be compiled as stand-alone executables? Did the development environment include a debugger? As NAG considered some of these questions, additional requirements began to form regarding a tool's flexibility and the power of its feature set.

## 5. Our Make versus Buy Decision

Based on the above requirements, only one tool came close to meeting our needs. It now became a decision to either buy this tool or make our own.

Cost was the initial differentiating factor. At the time, NAG employed some about 370 software testers. With our requirement that each tester be provided with a personal copy of any test software that NAG purchased, it was obvious that it would need to procure a site license of whatever tool it chose to use. The one tool that NAG considered buying was offered to us at $400,000 for a site license that capped at 400 users. It was NAG's estimation at the time that it could develop its own technology for a little over $100,000. Through negotiation, the price offered by the tool vendor was dropped to $200,000. This was tempting, but as a result of our research into test tools, cost was slowly becoming a secondary issue at best.

The flexibility of the test tool was now becoming an issue of more importance. Most of the tools that NAG looked at, including the only one that it considered buying, employed proprietary scripting languages. These languages for the most part were void of advanced data structures such as pointers and structures. NAG felt it best to have a tool built upon the strength of an established programming language. The authors envisioned a GUI test API that would be nothing more than a library that could be linked into a language such as C or C++. NAG certainly liked the idea of being able to compile test executables for both speed and ease of dispatching across a network. One of the vendors offered a GUI test C library, but only for the Windows environment. Management spoke to two other vendors about providing C language bindings to their cross-platform GUI test API's. Neither vendors seemed willing to talk seriously about this prospect. It was clear if NAG wanted a cross-platform GUI test library it would have to develop it in-house.

In the end, it was extensibility above all that became the most important issue in deciding to develop our own tool. The tool needed to allow the creation of "extensions" to test custom or client-drawn objects. Of the few tools that offered any solution at all, their solution was cumbersome and limited. If NAG owned its own technology, and that technology were nothing more that a GUI test library, it could easily extend more functionality to the tool by hooking into the application under test. Unlike our vendor friends, NAG had access to the code of the application that would be tested by this tool, that application was virtually a white box to us. In addition to a tighter integration between test tool and application under test, the thought of integrating the test tool into our established bug tracking system (which has a C API) was tempting. Certainly the most simple form of extensibility would be the ability to link a vast array of existing libraries into a language such as C++. Most commercial tools provided limited if any access to libraries outside of the tools intrinsic set of functions.

At this point it became clear that NAG was going to develop its own tool. Clarification of the issues discussed earlier led to specific technical requirements about the test API itself.

# 6. Technical Requirements

## 6.1. Removal of Context From Test Function Name

Test functions need to describe general test actions such as selecting a button or querying the state of a check box. Novell felt that a function that performs a specific action should be able to act on a variety of contexts. For example, the same function that performs a selection on a push button should also perform a selection on a check box or menu.

During our research, Novell noticed that other automated testing API's consisted of several functions performing the same essential action on different contexts. For example there would be functions such as CheckSelect, ButtonSelect, MenuSelect.

Having only one function performing a single action upon a variety of contexts is significant because it allows the user to remember only one function name for a given action. It also decreases the number of functions that must be maintained.

The context that a function acts upon is determined by an argument called a "context map". It consists of a character array which logically describes the object that receives the action. The following example shows how a single function in AppTester is used to select multiple objects:

```
Select("/WordPerfect*/@Menu/Format/Font");
Select("Font/Bold");
Select("Font/OK");
```

The above example turns on bold in WordPerfect. The example shows the Select function being used with a context map for a menu item, a check box, and a button. Note that the context map is a hierarchical description of a GUI object, starting from the desktop (denoted by the initial "/"), and moving to smaller objects. The context maps shown here use the object's name to describe the object being selected. It is also possible to create context maps which use the object's class name (such as [Button]) or its ordinal value, such as #1 or a combination of all three, such as "Font/[Button]#1", which would select the first button in the Font dialog.

Specifying the context in the context map instead of in the function name potentially reduces the amount of test script maintenance required if the context of a test is changed. For example, our Windows-based word processor uses a group of check boxes to display a series of font options. That same set of options might well be implemented as radio buttons on a UNIX version. Using the same function call for both means that modification of the script can be avoided. The only thing that differs is the context map, which can be stored in a resource file.

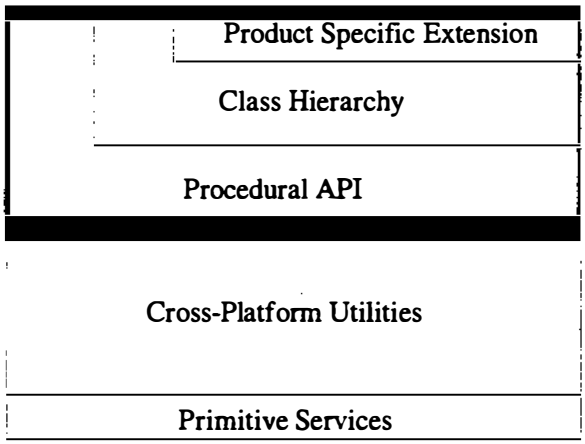
## 6.2. Test Tool Extensibility

Developing a test tool in-house provided the capability of testing custom objects created by NAG. The AppTester development team was able to use hooks provided by NAG developers which permitted AppTester to query and control NAG's custom objects. This was a beneficial solution for meeting the needs of the NAG testing groups as a whole, but wasn't efficient when

only one group required support for a custom control. These particular requests would normally get a lower priority than requests affecting all of the testing groups.

One way to provide quick support for the less common custom objects was to provide a means whereby the testers could extend the functionality of AppTester and customize it for their particular needs. This was accomplished by adding an additional interface to the AppTester API. This interface was an object oriented class hierarchy which gave the testing teams the same functionality found in the procedural API, but gave it in the form of objects. The class hierarchy consisted of classes matching each of the various windows appearing in GUI environments. The methods defined for each class corresponded to a set of procedural API calls that would be valid for any window of that class.

Deriving new classes from the class hierarchy and overriding the default class methods permitted testing groups to add support for custom objects without waiting for the AppTester group to implement it internally. If a newly derived class were to be needed by other testing groups, it could be easily shared until the AppTester development team implemented it internally.




## 6.3. Test Function Synchronization

In order for a GUI to keep up with a test script, Novell implemented synchronization into each test function. Our goal was to permit the same test script to run on a variety of machines without the need of placing static wait statements between commands.

To synchronize a script with the GUI, each function checks the context (specified by the context map) to see if it exists. If it does not exist, then the function yields momentarily to the system and than checks again. It will keep checking for a predetermined (adjustable) period of time until either the context appears or the default timeout period is exceeded in which case the function returns an error.

For example, a recent script used to test one of our apps was written on a 486/33. It was taken to Microsoft to test under Windows 95 and ran without incident on a P90. The test suite took 25 minutes on a 486/33 and six minutes on the P90. Solving the synchronization issue has bought NAG time both in the performance of the scripts and in the decreased maintenance that automatic synchronization allows.

### 6.4. Test Tool Intrusiveness

Developing the tool in-house gave NAG the advantage of knowing and controlling how intrusive the test functions would be. The intrusiveness of other tools was an unknown to NAG. Our original goal was to make each test function simulate the user as much as possible. When a test function performs an action, it performs mouse clicks or keyboard events rather than making a system call, thus circumventing actions that a user would perform. Even though as a default the test tool simulates a user's actions as much as possible, testers are able to select the level of intrusiveness.

### 6.5. Balance Between Flexibility and Ease of Use

The level of skill in using C or C++ varies widely among Novell testers. Therefore, AppTester needed to provide ease of use without sacrificing the power and flexibility that is available in C or C++. To accomplish this, Novell took advantage of C++ constructs to provide a layer of abstraction for testers. Novell created convenience classes that maintain strings, date/time data, files, etc.

### 6.6. Low-maintenance Code

Due to the size of the staff (usually 2 to 3 people) as well as the cross platform capabilities, developers were forced to write an API written on top of a platform-specific layers. A primitive services layer is defined where system specific functions are called. This primitive layer takes up approximately 20 percent of the total source code. This model has increased the maintainability of the test tool and allows NAG to add platforms in the future.

Currently, two people are assigned to both maintain the existing versions of AppTester and to port it to other platforms. These two people also provide training and documentation for the test tool.

### 6.7. Recorder

Providing a script recorder was not initially a high priority in the AppTester development process. Novell testing management felt it was necessary to have complete scripts written as soon as development code is complete, and a recorder was only useful after an application was written. However, after completion of the tool, the development team received many requests to provide one. Recorders are not capable of providing sophisticated, low maintenance scripts. They do

provide a framework for the tester to build upon. Also, it provides a means whereby users can learn about which functions to use and how to use them.

## 7. Other Benefits

### 7.1. Scheduling Flexibility

Other benefits have come from owning the GUI test tool. One is that enhancements to the tool can be done on NAG's own schedule. If an enhancement is important enough, resources can be shifted from other task to provide the necessary functionality. An example of this are tabbed dialogs in Windows 95. At the time of this writing, none of the tools Novell had originally evaluated had even been ported to Windows 95 and made commercially available. AppTester has been ported to 32-bit Windows for three months. When it became obvious that testers needed the ability to select tabbed dialogs, NAG testing managers and the developers of AppTester balanced this against other work the developers had to do. The decision was made to include the functionality. This change was available to testers within three weeks of the original request.

### 7.2. C++ API

Since AppTester is a C++ library, it can be integrated into a number of different environments and paradigms. This allows testers with varying backgrounds and degrees of aptitude to use the tool.

To date, the tool has been used in C, in two differing object-oriented C++ test harnesses, in AppBuilder for Windows, and with a recording tool. There are other possibilities. Fore example, it would be possible to make the API available under Tcl (tool command language).

## 8. Conclusion

From the decision in February 1994, it took three engineers one year to have a cross-platform tool in place on three platforms. Since that time, we have added a fourth platform, Windows '95.

Even though the Novell Applications Group is now deeply committed to the use of AppTester at the Novell Applications Group, those involved in its development occasionally wonder a few things: What will be the long term costs? Will the same amount of resources be needed to maintain and enhance the tool a year from now? Will AppTester need to be ported to other, newer platforms in a year or two? Novell does not have the answers to these questions. However, at the very least, Novell can take comfort in the fact that decisions about providing resources to maintain and improve the tool in the future will be Novell's alone to make.

While not every company will come out with the same decision Novell did during a make-or-buy analysis of GUI test tools, the need for a clear picture what the tool must do is important regardless of the outcome. It is hoped that this presentation sheds some light on perhaps otherwise unknown requirements.

# 9. Appendix 1 - Matrix of Functionality

## 9.1. Programming Language

### 9.1.1. Identifiers

| Identifier | A | B | C | D | E |
|---|---|---|---|---|---|
| Constant | ✔ | ✔ | ✔ | | ✔ |
| Variable | ✔ | ✔ | ✔ | ✔ | ✔ |

### 9.1.2. Atomic Data Types

| Data Type | A | B | C | D | E |
|---|---|---|---|---|---|
| Integer (Word) | ✔ | ✔ | | ✔ | ✔ |
| Long Integer (Long) | ✔ | ✔ | | | |
| Character (Byte) | ✔ | | | | |
| Floating Point (Float) | ✔ | ✔ | | | ✔ |
| Double-Precision Floating Point (Double) | ✔ | ✔ | | | |
| Number | | | ✔ | | ✔ |

### 9.1.3. User-defined Data Types

| Data Type | A | B | C | D | E |
|---|---|---|---|---|---|
| Alias Declaration/Type Definition | ✔ | ✔ | | | ✔ |
| Enumerated Type | ✔ | | | | ✔ |

### 9.1.4. Data Constructions

| Data Construction | A | B | C | D | E |
|---|---|---|---|---|---|
| Array | ✔ | ✔ | ✔ | ✔ | ✔ |
| String (Fixed Length) | ✔ | ✔ | | | |
| String (Variable Length) | ✔ | ✔ | ✔ | ✔ | ✔ |
| Pointer | ✔ | ✔ | | | |

| Data Construction | A | B | C | D | E |
|---|---|---|---|---|---|
| Structure/Record | ✔ | ✔ | | ✔ | ✔ |

### 9.1.5. Control Constructions

| Control Construction | A | B | C | D | E |
|---|---|---|---|---|---|
| IF-THEN-ELSE Statement | ✔ | ✔ | ✔ | ✔ | ✔ |
| FOR Loop | ✔ | ✔ | ✔ | ✔ | ✔ |
| WHILE Loop | ✔ | ✔ | ✔ | ✔ | ✔ |
| DO-WHILE Loop | ✔ | ✔ | ✔ | | |
| CASE Statement | ✔ | ✔ | ✔ | | ✔ |
| BREAK Statement | ✔ | ✔ | ✔ | | ✔ |
| CONTINUE Statement | ✔ | | ✔ | | ✔ |

### 9.1.6. Operators

#### 9.1.6.1. Assignment

| Operator | A | B | C | D | E |
|---|---|---|---|---|---|
| assign value (=) | ✔ | ✔ | ✔ | ✔ | ✔ |

#### 9.1.6.2. Arithmetic

| Operator | A | B | C | D | E |
|---|---|---|---|---|---|
| addition (+) | ✔ | ✔ | ✔ | ✔ | ✔ |
| subtraction (-) | ✔ | ✔ | ✔ | ✔ | ✔ |
| multiplication (*) | ✔ | ✔ | ✔ | ✔ | ✔ |
| division (/) | ✔ | ✔ | ✔ | ✔ | ✔ |
| modulo (%) | ✔ | ✔ | ✔ | ✔ | ✔ |
| exponentiation ($X^y$) | ✔ | | ✔ | | ✔ |

#### 9.1.6.3. Relational

| Operator | A | B | C | D | E |
|---|---|---|---|---|---|
| less than (<) | ✔ | ✔ | ✔ | ✔ | ✔ |
| greater than (>) | ✔ | ✔ | ✔ | ✔ | ✔ |
| less than or equal (<=) | ✔ | ✔ | ✔ | ✔ | ✔ |
| greater than or equal (>=) | ✔ | ✔ | ✔ | ✔ | ✔ |
| equal (==) | ✔ | ✔ | ✔ | ✔ | ✔ |
| not equal (!=) | ✔ | ✔ | ✔ | ✔ | ✔ |

#### 9.1.6.4. Logical

| Operator | A | B | C | D | E |
|---|---|---|---|---|---|
| AND | ✔ | ✔ | ✔ | ✔ | ✔ |
| OR | ✔ | ✔ | ✔ | ✔ | ✔ |
| NOT | ✔ | ✔ | ✔ | ✔ | ✔ |

#### 9.1.6.5. Bitwise

| Operator | A | B | C | D | E |
|---|---|---|---|---|---|
| AND | ✔ | ✔ | | ✔ | |
| inclusive OR | ✔ | ✔ | | ✔ | |
| exclusive OR | ✔ | ✔ | | ✔ | |
| complement (NOT) | ✔ | ✔ | | ✔ | |
| right shift | ✔ | | | ✔ | |
| left shift | ✔ | | | ✔ | |

### 9.1.7. Function Formats

| Simple Function | A | B | C | D | E |
|---|---|---|---|---|---|
| Simple Function | ✔ | ✔ | ✔ | ✔ | ✔ |
| Passing Arguments to a Function by Value | ✔ | ✔ | ✔ | ✔ | ✔ |

| Simple Function | A | B | C | D | E |
|---|---|---|---|---|---|
| Passing Arguments to a Function by Reference | | ✔ | ✔ | | ✔ |
| Passing Arguments to a Function by Pointer | ✔ | | | | |
| Returning an Integer Value from a Function | ✔ | ✔ | ✔ | ✔ | ✔ |
| Returning a Non-integer Value from a Function | ✔ | ✔ | ✔ | ✔ | ✔ |

## 9.2. GUI Test API

### 9.2.1. Function List

| | |
|---|---|
| AppExit | Unconditionally terminates a GUI application. |
| AppName | Obtains the name of an application module. |
| AppPath | Obtains the full path of the application module. |
| AppStart | Runs a GUI application. |
| AppWindow | Obtains the window map of the top level window of the application. |
| BmpCapture | Graphically captures a window. |
| BmpCaptureRect | Graphically captures a rectangle within a window.. |
| BmpCompare | Compares two graphic images that have been captured. |
| BmpGetCaptureCRC | Returns the CRC value of a specific bitmap. |
| BmpGetFileCRC | Returns the CRC value of a specific bitmap in a file. |
| ClassAddAlias | Adds or maps an alias (or custom) class to an existing base class. |
| ClassGetBase | Returns the base class number of a window. |
| ClassRemoveAlias | Removes an alias (or custom) class from the class alias table. |
| ClassResetTable | Resets the base/alias class table. |
| GetState | Returns the state of check boxes and radio buttons. Valid states are ON, OFF and HIGHLIGHTED. |

| | |
|---|---|
| IsChecked | Determines whether a check box or radio (option) button is checked. |
| IsDefault | Determines whether a push button is the default push button. |
| ItemDblSelect | Double selects (same as double clicking) a list box item. |
| ItemExists | Determines whether a list or menu item exists. |
| ItemExtendSelect | Extends the selection in an extend-selection list box. |
| ItemGetAccelerator | Obtains the accelerator of the menu item. |
| ItemGetIndex | Returns the index of the list box item, combo box list item or menu item referenced by name. |
| ItemGetSelIndex | Returns the index of the selected item in a list box or combo box. |
| ItemGetSelText | Obtains the text of the selected item in a list box or combo box. |
| ItemGetText | Obtains the text of the list item, combo box list item or menu item referenced by name. |
| ItemIsChecked | Determines whether a menu item is checked. |
| ItemIsEnabled | Determines whether a list box item or menu item is enabled. |
| ItemMouseClick | Generates a mouse click on a list box item. |
| ItemMouseDblClick | Generates a double mouse click on a list box item. |
| ItemMultiSelect | Selects an item in a multi-select list box. Previously selected items remain selected. |
| ItemMultiUnselect | Unselects an item in a multi-select list box. Previously selected items remain selected. |
| ItemSelect | Selects an item in a list box, combo box list or menu. |
| ItemTopIndex | Returns the index of the first visible item in a list box or combo box. |
| KeyPress | Generates a key press (but not release) of a specified key. |
| KeyRelease | Generates the release of a previously pressed key. |
| KeyType | Simulates keystrokes as if they were being typed at the keyboard. |
| ListGetItemCount | Returns the number of items in a combo box list, list box, or a menu. |
| ListGetMultiSelCount | Returns the number if items that are selected in a multi-selection or extend-selection list box. |

| | |
|---|---|
| ListGetMultiSelIndexes | Obtains a list of items that are selected in a multi-select list box. |
| ListIsExtendSelect | Determines whether a list box is extend-selection. |
| ListIsMultiSelect | Determines whether a list box is multi-selection. |
| MenuExists | Determines whether or not a menu exists. |
| MenuGetAccelerators | Obtains a lists of accelerators keys on a menu. |
| MenuGetDupMnemonics | Obtains the list of the ALT short-cut keys found more than once in the active window or menu. |
| MenuGetItemCount | Returns the number of items in a menu. |
| MenuGetMnemonics | Obtains the list of the ALT short-cut keys found in the active window or menu. |
| MouseButtonPress | Generates a mouse button press (but not release) of a specified mouse button. |
| MouseButtonRelease | Generates the release of a previously pressed mouse button. |
| MouseClick | Generates a single mouse click. |
| MouseClickXY | Generates a single mouse click at point (x,y). |
| MouseDblClick | Generates a double mouse click. |
| MouseDblClickXY | Generates a double mouse click at point (x,y). |
| MouseMultiClick | Generates a specified number of mouse clicks. |
| MouseMultiClickXY | Generates a specified number of mouse clicks at point (x,y). |
| ObjExists | Determines whether a specific GUI object exists. GUI objects include top level windows, child windows, buttons, boxes, etc. |
| ObjFind | Finds an object (window), specified by its name, and gives it focus. |
| ObjFindWait | Finds an object (window), specified by its name, and gives it focus. If the window does not exists, the function will check at regular intervals a the specified number of seconds until the window appears or returns an error. |
| ObjGetCaption | Returns the text of the object (window) caption. |
| ObjGetChildren | Obtains a list of children of a parent object (window). |
| ObjGetClass | Obtains the class name of an object (window). |

| | |
|---|---|
| ObjGetFocus | Returns the caption or ID of the object (window) that currently has input focus. |
| ObjGetID | Returns the ID (or handle) of an object (window) specified by its caption. |
| ObjGetIndex | Returns the index (order) of an object (window) relative to other sibling objects (windows). |
| ObjGetMap | Obtains a hierarchal reference of an object (window) specified by its ID. This reference is called the widow map. |
| ObjGetParent | Obtains the caption or ID for the parent object (window) of a specified object (window). |
| ObjGetPos | Obtains the screen origin of an object (window). |
| ObjGetRect | Obtains the location and size of an object (window). |
| ObjGetSize | Obtains the size (width, height) of a object (window). |
| ObjHasFocus | Determines whether a control (object) has input focus. |
| ObjIsEnabled | Determines whether a control (object) is enabled. |
| ObjIsVisible | Deterimines whether an object (window) is visible. |
| ObjSetFocus | Sets the input focus to a specified object (window). |
| PointerMove | Moves mouse pointer to a point within an object (window). |
| ScrollByLine | Scrolls the scroll bar by the specified number of lines. |
| ScrollByPage | Scrolls the scroll bar by the specified number of pages. |
| ScrollGetPos | Obtains the position of a scroll bar. |
| ScrollGetRange | Returns the range of a scroll bar. |
| ScrollToMax | Scrolls the scroll bar to the maximum position. |
| ScrollToMin | Scrolls the scroll bar to the minimum position. |
| ScrollToPos | Scrolls the scroll bar to the specified position. |
| Select (Toggle) | Selects a control (object). GUI objects that can be selected by this function include check boxes, push buttons, and option buttons. This funtions will essentially toggle the state of a check box. |
| SetState | Sets the state of check boxes and radio buttons. Valid states are ON, OFF and TOGGLE. |

| | |
|---|---|
| TextClear | Clears the text from an edit field. |
| TextGetContents | Obtains the text from an edit field. |
| TextGetFirstVisible | Returns the index of the first visible character in an edit field. |
| TextGetInsert | Returns the insertion point offset, relative to the beginning of an edit field. |
| TextGetMultiLineContents | Obtains the text of a line (specified by line number) in a multi-line edit field. |
| TextGetMultiLineCount | Returns the number of text lines in a multi-line edit field. |
| TextGetMultiLineInsert | Returns the insertion point offset, relative to the beginning of the current line in a multi-line edit field. |
| TextGetMultiLineNum | Returns the number of the current line (insertion point) in a multi-line edit field. |
| TextGetSelContents | Obtains the selected text in an edit field. |
| TextGetSelIndexes | Obtains the beginning and ending indexes of a selection of text in an edit field. |
| TextInsert | Inserts text into an edit field. If text is selected prior to insertion, then the selected text is replaced. Otherwise the text is inserted into the edit field at the insertion point. |
| TextIsMultiLine | Determines whether an edit field is single or multi-line. |
| TextSetContents | Sets the text of an edit field. All existing text is replaced regardless any previous selction. |
| TextSetInsert | Sets the insertion point offset, relative to the beginning of an edit field. |
| TextSetMultiLineInsert | Sets the insertion point offset, relative to the beginning of the current line in a multi-line edit field. |
| TextSetSelIndexes | Sets the beginning and ending indexes of a selection of text in an edit field. |
| WndClose | Closes a top-level or MDI child window. |
| WndGetActive | Returns the caption or ID of the active window. |
| WndIsMax | Determines whether a window is maximized. |
| WndIsMin | Determines whether a window is minimized. |
| WndMaximize | Maximizes a window. |

| | |
|---|---|
| WndMinimize | Minimizes a window. |
| WndRestore | Restores a window that has been either minimized or maximized. |
| WndSetActive | Sets a window active and gives it input focus. |
| WndSetPos | Sets the position of a window. The coordinates are relative to its parent window. This means that an MDI Child window will be positioned relative to a program's main window. Other top level windows are positioned relative to the desktop. |
| WndSetRect | Sets the location and size of a top-level or MDI Child window. The coordinates are relative to its parent window. This means that an MDI Child window will be positioned relative to a program's main window. Other top level windows are positioned relative to the desktop. |
| WndSetSize | Sets the size (width, height) of a window. |

### 9.2.2. Application Functions

This section contains functions that allow a GUI application to be started from within a test script.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| AppExit | | | | | ✔ |
| AppName | | | ✔ | ✔ | ✔ |
| AppPath | | | | | |
| AppStart | ✔ | ✔ | ✔ | ✔ | ✔ |
| AppWindow | ✔ | | | ✔ | ✔ |

### 9.2.3. Bitmap Functions

This section contains functions that permit the creation and comparison of bitmap screen captures.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| BmpCapture | ✔ | ✔ | ✔ | | ✔ |
| BmpCaptureRect | ✔ | ✔ | ✔ | | ✔ |
| BmpCompare | ✔ | ✔ | ✔ | | ✔ |
| BmpGetCaptureCRC | | | | | ✔ |

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| BmpGetFileCRC | | | | | ✔ |

### 9.2.4. Class Functions

This section contains functions that allow the aliasing of object base classes to custom object classes. Once a base class has been aliased to a custom class, objects of a custom class have access to functions and services that query and manipulate objects of a predefined base class.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| ClassAddAlias | ✔ | ✔ | | | |
| ClassGetBase | ✔ | | | | |
| ClassRemoveAlias | ✔ | | | | |
| ClassResetTable | ✔ | ✔ | | | |

### 9.2.5. Object Functions

This section contains functions that query and manipulate GUI objects. In this context, objects include top-level windows, MDI Client windows, list boxes, combo boxes, edit fields, static text fields, check buttons, radio (option) buttons and push buttons.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| KeyPress | | ✔ | ✔ | ✔ | |
| KeyRelease | | ✔ | ✔ | ✔ | |
| KeyType | ✔ | ✔ | ✔ | ✔ | |
| MouseButtonPress | | ✔ | ✔ | ✔ | |
| MouseButtonRelease | | ✔ | ✔ | ✔ | |
| MouseClick | | ✔ | ✔ | ✔ | ✔ |
| MouseClickXY | ✔ | ✔ | | | ✔ |
| MouseDblClick | | ✔ | ✔ | ✔ | ✔ |
| MouseDblClickXY | ✔ | ✔ | | | ✔ |
| MouseMultiClick | | | ✔ | | ✔ |
| MouseMultiClickXY | | | | | ✔ |

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| ObjExists | ✔ | ✔ | ✔ | ✔ | ✔ |
| ObjFind | | ✔ | | ✔ | |
| ObjFindWait | | ✔ | | ✔ | |
| ObjGetCaption | ✔ | ✔ | ✔ | ✔ | ✔ |
| ObjGetChildren | ✔ | | | ✔ | ✔ |
| ObjGetClass | ✔ | ✔ | ✔ | ✔ | ✔ |
| ObjGetFocus | ✔ | | | | ✔ |
| ObjGetID | ✔ | | ✔ | | ✔ |
| ObjGetIndex | ✔ | | | ✔ | ✔ |
| ObjGetMap | | | | | ✔ |
| ObjGetParent | ✔ | ✔ | | | ✔ |
| ObjGetPos | | ✔ | ✔ | ✔ | |
| ObjGetRect | ✔ | ✔ | ✔ | ✔ | ✔ |
| ObjGetSize | | ✔ | ✔ | | |
| ObjHasFocus | ✔ | ✔ | ✔ | | |
| ObjIsEnabled | ✔ | ✔ | ✔ | | ✔ |
| ObjIsVisible | ✔ | ✔ | ✔ | | |
| ObjSetFocus | ✔ | ✔ | | | ✔ |
| PointerMove | | | ✔ | ✔ | ✔ |

### 9.2.6. Client-Window Functions

This section contains functions that query and manipulate top-level and MDI Client windows. These are windows that usually contain a frame that allow the window to be moved, resized, minimized, maximized and closed.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| MenuGetDupMnemonics | | ✔ | | | |
| MenuGetMnemonics | | ✔ | | | |

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| WndClose | | | ✔ | ✔ | ✔ |
| WndGetActive | ✔ | ✔ | | ✔ | ✔ |
| WndIsMax | ✔ | ✔ | | | |
| WndIsMin | ✔ | ✔ | | | |
| WndMaximize | | ✔ | ✔ | ✔ | ✔ |
| WndMinimize | | ✔ | ✔ | ✔ | ✔ |
| WndRestore | | ✔ | ✔ | ✔ | ✔ |
| WndSetActive | ✔ | ✔ | ✔ | ✔ | ✔ |
| WndSetPos | | ✔ | ✔ | ✔ | ✔ |
| WndSetRect | | ✔ | | | |
| WndSetSize | | ✔ | ✔ | ✔ | ✔ |

### 9.2.7. Menu Functions

This section contains functions that query and manipulate items within a special resource of a window called a menu.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| ItemExists | ✔ | ✔ | | ✔ | ✔ |
| ItemGetAccelerator | | | | ✔ | |
| ItemGetIndex | | | ✔ | ✔ | |
| ItemGetText | ✔ | ✔ | ✔ | ✔ | |
| ItemIsChecked | ✔ | ✔ | | | ✔ |
| ItemIsEnabled | ✔ | ✔ | ✔ | | ✔ |
| ItemSelect | ✔ | ✔ | ✔ | ✔ | ✔ |
| MenuExists | ✔ | | | ✔ | |
| MenuGetAccelerators | | | | | |
| MenuGetDupMnemonics | | ✔ | | | |
| MenuGetItemCount | ✔ | ✔ | ✔ | | ✔ |

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| MenuGetMnemonics | | ✔ | | | |

### 9.2.8. Check Box Functions

This section contains functions that query and manipulate check box objects.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| GetState | ✔ | ✔ | ✔ | ✔ | ✔ |
| IsChecked | ✔ | | | | |
| Select (Toggle) | ✔ | ✔ | ✔ | ✔ | ✔ |
| SetState | | | ✔ | | ✔ |

### 9.2.9. Edit Field Functions

This section contains functions that query and manipulate edit field objects. Many of these functions apply to combo box edit fields.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| TextClear | ✔ | | | | ✔ |
| TextGetContents | ✔ | ✔ | ✔ | ✔ | ✔ |
| TextGetFirstVisible | | ✔ | | | |
| TextGetInsert | ✔ | ✔ | | | |
| TextGetMultiLineContents | ✔ | ✔ | ✔ | ✔ | ✔ |
| TextGetMultiLineCount | | ✔ | ✔ | | |
| TextGetMultiLineInsert | | ✔ | ✔ | | |
| TextGetMultiLineNum | | ✔ | | | |
| TextGetSelContents | ✔ | ✔ | | | ✔ |
| TextGetSelIndexes | ✔ | ✔ | ✔ | | ✔ |
| TextInsert | ✔ | | ✔ | | |
| TextIsMultiLine | | | | | ✔ |
| TextSetContents | ✔ | ✔ | ✔ | | ✔ |

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| TextSetInsert | ✔ | | ✔ | | |
| TextSetMultiLineInsert | ✔ | | ✔ | | |
| TextSetSelIndexes | ✔ | ✔ | ✔ | | |

### 9.2.10. List Box Functions

This section contains functions that query and manipulate list box objects and list items. Most of these functions apply to the list portion of a combo box.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| ItemDblSelect | ✔ | ✔ | ✔ | | ✔ |
| ItemExists | ✔ | ✔ | | | ✔ |
| ItemExtendSelect | ✔ | ✔ | ✔ | | ✔ |
| ItemGetIndex | ✔ | | | | |
| ItemGetSelIndex | ✔ | ✔ | ✔ | | ✔ |
| ItemGetSelText | ✔ | ✔ | ✔ | | ✔ |
| ItemGetText | ✔ | ✔ | ✔ | | |
| ItemMouseClick | ✔ | ✔ | | | ✔ |
| ItemMouseDblClick | ✔ | ✔ | | | ✔ |
| ItemMultiSelect | ✔ | ✔ | ✔ | | ✔ |
| ItemMultiUnSelect | ✔ | ✔ | | | ✔ |
| ItemSelect | ✔ | ✔ | ✔ | | ✔ |
| ItemTopIndex | | ✔ | | | |
| ListGetItemCount | ✔ | ✔ | ✔ | | ✔ |
| ListGetMultiSelCount | | ✔ | | | |
| ListGetMultiSelIndexes | | ✔ | ✔ | | ✔ |
| ListIsExtendSelect | ✔ | | | | ✔ |
| ListIsMultiSelect | ✔ | | | | ✔ |

### 9.2.11. Push Button Functions

This section contains functions that query and manipulate push button objects.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| IsDefault | ✔ | ✔ | | | ✔ |
| Select | ✔ | ✔ | ✔ | ✔ | ✔ |

### 9.2.12. Radio (Option) Button Functions

This section contains functions that query and manipulate radio button objects.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| GetState | ✔ | ✔ | ✔ | ✔ | ✔ |
| IsChecked | ✔ | | | | |
| Select | ✔ | ✔ | ✔ | ✔ | ✔ |
| SetState | | | ✔ | | |

### 9.2.13. Scroll Bar Functions

This section contains functions that query and manipulate scroll bar objects.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| ScrollByLine | ✔ | | ✔ | | ✔ |
| ScrollByPage | ✔ | | ✔ | | ✔ |
| ScrollGetPos | ✔ | | ✔ | ✔ | ✔ |
| ScrollGetRange | ✔ | | | ✔ | ✔ |
| ScrollToMax | ✔ | | ✔ | ✔ | ✔ |
| ScrollToMin | ✔ | | ✔ | ✔ | ✔ |
| ScrollToPos | ✔ | | ✔ | ✔ | ✔ |

### 9.2.14. Static Text Functions

This section contains functions that query static text field objects.

| API Function | A | B | C | D | E |
|---|---|---|---|---|---|
| TextGetContents | ✔ | ✔ | ✔ | ✔ | ✔ |

# 10. Appendix 2 - Sample Script

```
/*============================================================================
      SOURCE FILE

      File:       notepd01.cpp
      Title:      A simple notepad test using the WP AppTester
      Desc:       This particular script performs a simple NotePad test.
                  It executes Window's Notepad and loads your autoexec.bat
                  file.
      Owner:      Any Owner
      Tabs:       Every 4

      Copyright (C) 1994, 1995 WordPerfect Corp., Novell Inc., All Rights
      Reserverd

----------------------------------------------------------------------------
      Revision History: (Most recent change on top)

      Name                          Date              Description

      Any User                      Feb 7, 94    Creation

============================================================================*/


/*============================================================================
      Defines
============================================================================*/


/*============================================================================
      Include files
============================================================================*/

#include <apptest.h>

/*============================================================================
      Prototypes
============================================================================*/


/*============================================================================
      Function:   TestMain
----------------------------------------------------------------------------
      Desc:       Main test function.  All WP Test API scripts have a TestMain
                  function.  This particular function performs a simple
                  NotePad test. It executes Window's Notepad and loads
                  your autoexec.bat file.
      In:         Command line paramters (if any).
                  These are not used in this test.
      Out:        none
      Ret:        User-defined error value.
      Notes:      none
```

```
==========================================================================*/
#pragma argsused
unsigned TestMain(unsigned _argc, char *_argv[])
{

      // Start Notepad - if failure, return error
      if (!XpAppStart("notepad.exe", "/Notepad*"))
      {
            // Normally, you do not want to call a function that requires user
            // interaction when an error occurs.  This is done here,
            //however, to  simplify the sample script.
            MessageBox(NULL, WpGetErrStr(WpGetErrNum()), "WpAppStart Error",
                              MB_OK | MB_ICONSTOP);
            return ERR;
      }

      // Bring up file/open dialog
      if (XpSelect("/Notepad*/@Menu/File*/Open*") == ERR)
      {
            MessageBox(NULL, WpGetErrStr(WpGetErrNum()), "WpGetErrStr Error",
                              MB_OK | MB_ICONSTOP);
            return ERR;
      }

      // Type autoexec.bat in the File name field, then <enter>
      if (XpTypeKeys("Open*", "c:\\autoexec.bat<enter>") == ERR)
      {
            MessageBox(NULL, WpGetErrStr(WpGetErrNum()), "WpTypeKeys Error",
                              MB_OK | MB_ICONSTOP);
            return ERR;
      }

      return 0;
}

/*==========================================================================
      END OF SOURCE FILE
==========================================================================*/
```

# Writing GUI Specification in C

Shankar Chakrabarti, Hewlett-Packard Company

## Abstract

*Formal specification of GUI-driven application is so difficult that it is hardly ever attempted in a production environment. To date we know of no methodology to write GUI specification in a popular language like C. This talk will provide a short description of a tool called Synlib and will describe how a methodology to create tests for a GUI-driven application using Synlib yields a specification of the application in C. Such specifications—called executable specifications—can be created in the absence of a functional copy of the target application by executing the specification itself.*

## About the speaker

*Dr. Shankar Chakrabarti did his graduate work in computer science at Oregon State University and is currently an R&D engineer at Hewlett-Packard Company. He has been involved with the X-Windows effort from its inception. Dr. Chakrabarti has developed a series of successful tools and methodologies to test X/Motif-based software. His primary research interest is in the areas of software verification, testing paradigms, and programming languages.*

# Writing GUI specifications in C

Shankar L. Chakrabarti
Workstation Technology Group
Hewlett-Packard Company
1000 NE Circle Blvd., Corvallis, Or 97330
sankar@hp-pcd.cv.hp.com

In this talk we present an interesting perspective gained while developing automated tests for a graphical user interface (GUI) based system. We found that while trying to create tests for a GUI driven text editor using a non-conventional methodology and a C-language library called Synlib, we ended up creating an executable specifications for the editor.

**Testing of a GUI software is mostly a three step process**

1. Often the starting material is an informal textual description of the target application. This description is generated often in the beginning phase of the development cycle of the application. The stated purpose of this description is to let others know the intended "look and feel" and through that indirectly the intended behavior of the application under construction.
2. The second step starts when the application is nearly functional. In this stage, the test engineers, often with the help of the application developers, create sample "test cases". To our knowledge, there is no standard or accepted method of creating test cases. For GUI applications, each test case often describes, again informally, how to manipulate the components of the target GUI with keyboard and pointer devices. It also may describe what an user is likely to see in the course of these manipulations if the the application under test were to behave "properly". Sometime the test cases are described in the form of tables which can be checked off in step 3.
3. The third step is the actual testing step. In manual testing, testers use the "test cases" (step 2) as a guide to interact with the application to be tested. The tester watches the response of the application as it accepts and processes the inputs delivered to it. The test cases also guide the tester to decide whether the application was behaving properly or not. In automated testing, recorder programs are used to capture the tester's interactions with the application. Recording tools are also capable of capturing specific features of the application at various stages of recording. Automation is achieved when the recorded interactions are played back to the application by special "Player" programs. The applications accept the inputs from the Player and respond to them as they did

during recording. The difference is that no human intervention is needed during playback. Human assistance is often necessary to analyze the results of playback.

We are investigating a testing methodology in which the recording step is completely eliminated. Instead of the tester manually interacting with the application as described in the test cases, in our method, the test engineer uses the informal text descriptions ( step 1) and a C-language library called Synlib to write C-language programs to describe the behavior of the target application. The C-language programs thus created are directly executed on the target applications to determine if the application displays behavior coded in these programs.

The C-language library Synlib, is an integral part of this process. Various aspects of this library have been published elsewhere. The talk will provide a short summary of this library. This library provides capabilities:
(a) to name objects of interest within a GUI;
(b) to deliver desired keyboard and/or pointer inputs to named objects within GUI of interest;
(c) to verify if the GUI is in a desired state.

Using these capabilities we have written a set of C-language programs which can be used to test a GUI driven text editor for a desktop environment. Each of these programs are crafted to test one or a few specific properties of the GUI text editor. Collectively the set of programs comprised a suite to test and verify the GUI text editor. It was possible to create this test suite without having to record anything at all.

**Test Or Specification**

On examining the test programs thus created, we found that the programs themselves had no dependence on the internal construction of target application. The test programs do not refer to the internal widget hierarchy; nor do they refer to the names or types of widgets used to construct the target application. The objects of interest in a GUI are named by a Synlib construct called FocusMap - which is a mechanism to formally specify the logical organization of user accessible objects within a GUI window. The test programs are merely descriptions of properties of objects available to the user for manipulation. The property of an object is expressed by stating what should happen when a defined input is delivered to a named object at a given state of the application. The net result of this methodology is that Synlib based test programs describe the properties of an application independent of any particular

implementation. And since the test programs describe the properties using constructs of a formal language ( "C"), we argue that these programs represent a formal model of behavior of the target GUI application. Moreover since our experience shows that these test programs can be created even in absence of a functional copy of the target application, we venture to suggest that each such Synlib based test program is in effect, a formal, albeit partial, specification of the target GUI application. And since these programs can be directly executed to verify if the target GUI application complies with the expectations coded in the specification, these Synlib based test programs may be also be termed "executable specifications".

In this exercise we started with the informal textual description and as we were developing individual test programs we realized that following our methodology, we were actually converting an informal description to a formal specification of the intended application. In the talk we will trace several examples to show how we came about this realization.

The concept of specification-based testing and executable specifications are not entirely new. We think however that our work has two interesting aspects worth mentioning. Most of the work on specification-based testing is related to process-based or model-based specification languages. To our knowledge the concept of writing specifications in a popular language like C is quite novel. Secondly in other systems, specification and test are two separate things; tests are often derived from the specification. In contrast in our case specification is the test itself. Finally, to our knowledge formal methods to specify GUI applications or other non-toy programs have been largely unsuccessful. Whenever attempted the formal methods are found to be overtly complex and are quickly abandoned after initial effort. Perhaps at the current stage of software engineering, the formal methods are too complex and promise little, if any, payback in a production environment. In contrast the methodology described here are found to be simple, intuitive and are easily adapted and propagated in a production environment.

---

***** The author gratefully recognizes the active and invaluable collaboration of Mr. Sami Mohammed ( sami@x.org) and Mr. Rajeev Pandey ( rpandey@hp_pcd.cv.hp.com) in many aspects of this work.

# TRUBAC: Testing Expert Systems with Rule-Base Coverage Measures

Valerie Barr*
Department of Computer Science
Hofstra University

In this paper we present a tool, TRUBAC (Testing with RUle-BAse Coverage), which implements a new approach to verification and validation of rule-based expert systems. In TRUBAC we use a set of rule-base coverage measures to evaluate and guide the dynamic testing (validation) of a rule-base. In addition, the rule-base representation used by TRUBAC for validation can also be used in a very straightforward fashion to carry out verification as well, allowing one method to serve both jobs.

TRUBAC implements an approach to the verification and validation of rule-based systems which is based on the data flow approach to testing of procedural programs. In the paper we begin with an overview of the process carried out by TRUBAC. We then introduce the rule-base representation used by TRUBAC, followed by a discussion of the rule-base coverage measures, and dynamic and static analysis in TRUBAC. We end with a number of case studies in which TRUBAC is used to evaluate the testing of a number of rule-based expert systems developed in different rule-base languages.

Valerie Barr is near completion of her PhD in Computer Science at Rutgers University, and is an Assistant Professor of Computer Science at Hofstra University. She received her MS in CS in 1979 from New York University, followed by 10 years in industry before her return to graduate study in 1989. In addition to research in software testing, specifically the testing of expert systems, she is also interested in the developing area of medical informatics.

*Department of Computer Science, Hofstra University, Hempstead, NY 11550 Email: vbarr@cs.rutgers.edu

# TRUBAC: Testing Expert Systems with Rule-Base Coverage Measures

Valerie Barr*
Department of Computer Science
Hofstra University

## 1 Introduction

In this paper we present a tool, TRUBAC (Testing with RUle-BAse Coverage), which implements a new approach to verification and validation of rule-based expert systems. An expert system contains an encoding of an expert's knowledge about some problem domain. In a rule-based system the knowledge is stored in a series of rules, usually in an if-then format. (Readers unfamiliar with expert systems may wish to look at Appendix A for a introduction to rule-based systems). For example, an expert system that diagnoses why a car does not start might have the rule

**if temperature between 0 and 50 and the starter cranks slowly**
**then the choke may be stuck.**

Increasingly expert systems are being developed for a wide range of applications, such as medical diagnosis and manufacturing control. However the quality of the answers provided by an expert system depends on the degree to which the knowledge stored within the expert system is correct.

The usual method for dynamic test of a rule-based expert system is to provide a number of test cases with known answers, comparing the system's answers with the expected answers, thereby obtaining a measure of how accurately the rule-base performs on the known test cases. The weakness of this mode of testing is that there is no indication of the extent to which the rule-base is actually exercised by the test cases. In fact, it is possible that there may be sections of the rule-base which were not used at all in the evaluation of the test cases, in which case the accuracy figure obtained applies to only a segment of the rule-base.

In TRUBAC we use a set of rule-base coverage measures to evaluate and guide the dynamic testing (validation) of a rule-base. In addition, the rule-base representation used by TRUBAC for validation can also be used in a very straightforward fashion to carry out verification as well, allowing one method to serve both jobs.

The research area of knowledge base verification, validation and testing (VV&T) is replete with definitions of these terms. In this paper we use the following definitions:

- Validation – check that the system is behaving in accordance with the specification. We take this to mean "gives the right answer". Alternatively, we want to check that the conclusion of the system "resembles" that of the human expert(s) who provided knowledge for the system [Pre89].

*Department of Computer Science, Hofstra University, Hempstead, NY 11550 Email: vbarr@cs.rutgers.edu

- Verification – detect internal inconsistencies in a rule-base, such as redundancy, conflict, or cycles (when not allowed). Check that the system is logically sound and complete.

Verification can be carried out statically, by studying the rule-base without running test cases. However, validation must be a dynamic process whereby the rule-base is applied to test cases and the resulting conclusions are evaluated.

Finally, while existing VV&T tools are limited to rule-bases without dynamic computation of certainty factors, TRUBAC has been extended to evaluate rule-based expert systems which incorporate dynamic computation of certainty factors.

TRUBAC implements an approach to VV&T of rule-based systems which is based on the data flow approach to testing of procedural programs [RW85; FW85]. In this paper we begin with an overview of the process carried out by TRUBAC. We then introduce the rule-base representation used by TRUBAC, followed by a discussion of the rule-base coverage measures, and dynamic and static analysis in TRUBAC. We then present a number of case studies in which TRUBAC is used to evaluate the testing of a number of rule-based expert systems developed in different rule-base languages.

## 2 Overview of TRUBAC

TRUBAC begins a rule-base evaluation by constructing a directed acyclic graph (DAG) representation of the rule-base. During construction of the DAG redundant rules, simple contradictory rules and potential contradictions (ambiguities) are identified. After DAG construction is complete the user can run a static analysis of the rule-base. This will report on dangling conditions, useless conclusions, and cycles in the rule-base.

This is followed by dynamic analysis of the rule-base using test cases. It is in this area that TRUBAC differs the most from other expert system analysis tools. After each test case has been processed, the user can indicate which of several rule-base coverage measures (RBCM) should be reviewed in order to determine the quality of the test data supplied thus far, or the user can elect to run additional test cases. Presumably the user would start by providing sufficient test data to satisfy the simplest functional measure and proceed to the more difficult structural measures. Finally, if the user is not able to provide sufficient data to attain the desired degree of rule-base coverage, the user can elect to have TRUBAC provide synthesized data, which can then be reviewed by an expert to determine if the data represents a valid case.

## 3 Rule-base Representation

The representation used in TRUBAC is based on the AND/OR graph implicit in the rule-base [Mes91]. The representation is a DAG which contains a "source node" representing system input, a "sink node" with an edge to it from each system goal, and interior nodes which represent either antecedent components or disjunction or conjunction operators. In this framework we can easily represent rule-bases that encode reasoning processes that do not rely on dynamic computation of certainty factors, including those in which the certainty factors are hard coded within the definition of consequents. With simple modifications we can also represent rule-bases with dynamic computation of certainty factors.

For example, the representation of the rule

*If* P1 *&* P2 *then* R1

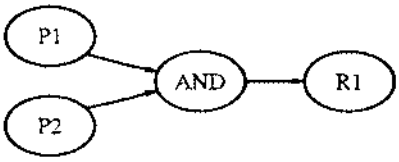



Figure 1: DAG representation of simple rule

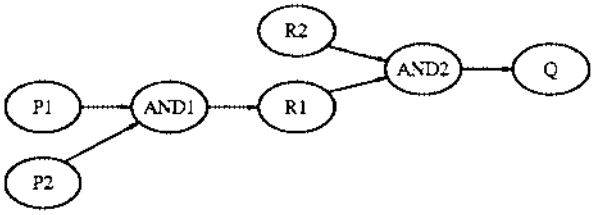



Figure 2: DAG representation of two related rules

is shown in Figure 1.

The complete graph of a rule-base is constructed by linking together the individual structures for successive rules. For instance, the two rules

*If* P1 *&* P2 *then* R1
*If* R1 *&* R2 *then* Q

would be represented by the sub-DAG shown in Figure 2.

The first step of DAG construction is creation of the *source* and *sink* nodes, in the following sense. The *source* node corresponds to data supplied by the user to the database, the source of facts supplied to the system during a consultation. The *sink* node corresponds to success in reaching one of the goals of the system. Then the rule-base is processed, starting with the lists of facts and goals. Each fact is entered into a hash table, as well as into the DAG. The graph node created for the fact will have the source as its parent, and will appear on the neighbor list of the source. Goals are not put into the DAG immediately, but instead are put onto a separate list. This allows us to determine, for each rule processed, if its consequent is a goal or an intermediate hypothesis. Furthermore, it makes it extremely simple to determine, after the DAG is constructed, if there are any goals for which no rules appear in the rule-base. We note that creation of this extra goal list does not require a significant amount of extra storage space, since the number of goals in a system tends to be very small, relative to the number of facts and intermediate hypotheses.

After the facts and goals have been processed, the rule base is processed, one rule at a time, and the interior components of the DAG are created. Rule order is unimportant. As each rule is processed, the following steps are carried out:

1. Parse the antecedent, constructing a graph representation of it. In this graph the facts and intermediate hypotheses appear as leaf nodes, and the AND, OR, NOT operators appear as interior nodes, with one as the root of the graph. For the antecedent of the rule

   if a or not (b or c) then d

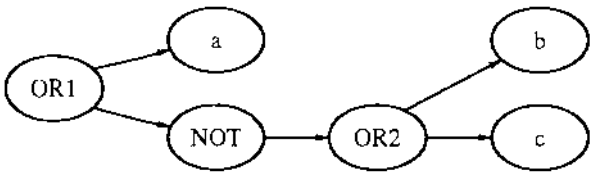



Figure 3: Graph of simple antecedent

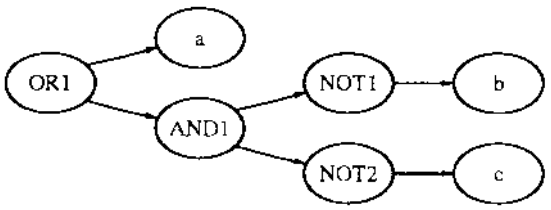



Figure 4: DeMorgan's Law applied to antecedent

this step will produce the graph shown in Figure 3.

2. Convert the antecedent to Conjunctive Normal Form (CNF). This is done by modifying the graph created in step 1, rather than working with the rule text directly. For our example rule, Figure 4 shows the graph after the first conversion step, with the **not** applied to its argument so that any **not** in the graph will have only one child. Figure 5 shows the graph which results from the completed conversion to CNF. Note that the node for the fact (or intermediate hypothesis) *a* has been replicated in the process of converting to CNF. We note that while, in general, conversion to CNF can cause a combinatorial explosion in the number of terms (precisely because of the sort of node replication that was necessary with node *a* in the example), we do not expect a problem in this application due to the reasonable size of antecedents (on the order of 5 terms).

3. Create a hash table entry for the full antecedent, in the form of an ordered list of conjuncts, in which each conjunct is also represented by an ordered list. Look up the antecedent in the hash table.

4. If the antecedent is already in the hash table then note that a redundancy check will have to be done

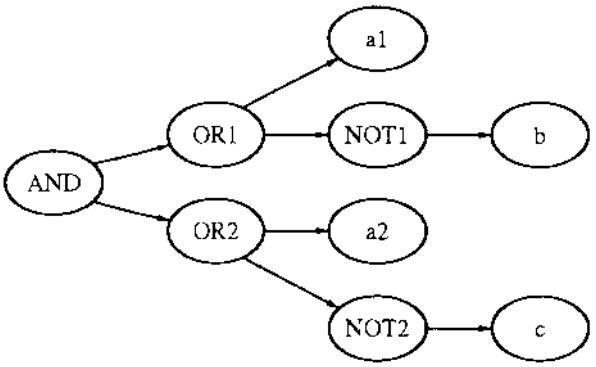



Figure 5: CNF form of graph of simple antecedent

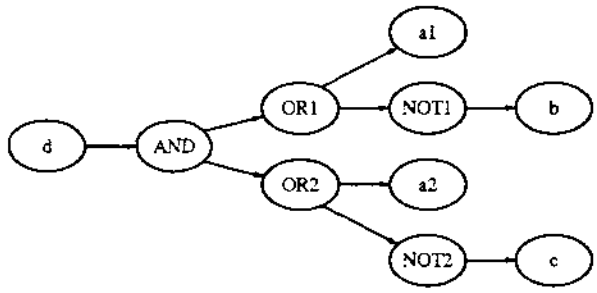



Figure 6: Graph of complete rule

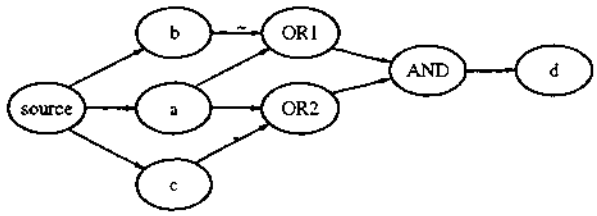



Figure 7: Graph of complete rule integrated into DAG

after the consequent is seen. Otherwise the antecedent is put into both the hash table and the DAG.

5. Get the consequent. If it is in the hash table already, then retrieve its graph location. Otherwise put it into both the hash table and the DAG, making appropriate connections to the sink node if the consequent is a goal.

6. Make the necessary graph connections between the antecedent and the consequent by making the consequent a parent of the antecedent and the antecedent a child of the consequent. We do this by using the root of the graph containing the CNF representation of the antecedent as the node to and from which connections to the consequent will be made. For our example rule, the graph with the consequent added is shown in Figure 6.

7. Up to this point the graph of the rule has actually been constructed as a binary tree, with each node having left and right children. Now we must rearrange the pointers to integrate the rule's graph into the larger rule-base DAG structure. We can think of this process as one in which the tree is essentially turned upside down. Basically, every child in the tree becomes a parent of its "tree parent", and every parent in the tree becomes a neighbor (child) of its "tree children". Furthermore, we eliminate all NOT nodes by moving negation information onto the appropriate edge. Finally, wherever a fact appears in the antecedent tree, we connect instead to the node for the fact which is already in the DAG. For our example rule, the relevant section of the final DAG is shown in Figure 7 (assuming that $a$, $b$, $c$ are all facts of the system, and are therefore connected to the source. Note that $\sim$ indicates negation on the graph edge).

# 4 Testing with Rule-Base Coverage Measures

In developing the testing approach used in TRUBAC, we have applied to rule-based systems the approach used in the data flow testing of procedural programs.

## 4.1 Data Flow Testing of Procedural Program

The main principle of data flow testing is: if there is a value assignment to a variable that is not used in some subsequent computation or predicate in the program during the program test, then we cannot be sure that the steps leading up to the assignment were correct. Data flow testing involves tracking input variables through a program, following them as they are modified, until they are ultimately used to produce output values. Ideally the program tester would like to provide as many different test cases as are necessary to "exercise" paths between all variable assignments and subsequent variable uses.

Rapps and Weyuker [RW85] developed a family of data flow testing criteria, the strongest of which requires that the test set include data which causes the program to execute every path connecting a variable definition with a subsequent use of that variable in a computation or a predicate. ASSET [FW85] is a software tool for Pascal programs which incorporates these data flow testing criteria, thereby evaluating the usefulness of test data and helping the user to select test data. The user provides a module to be tested, along with test data and the specified criterion. ASSET then tells the user which paths (as required by the chosen criterion) were not traversed by the given test data. The user can then provide additional test data until either all required paths are exercised or the user can see that some path may not be executable. In addition, ASSET serves as a debugging aid by indicating what path through the program the user should study if a particular test case gives an incorrect result.

## 4.2 Data Flow Applied to Rule-Bases

There are obvious analogies we can make between data flow testing and the testing of expert systems. The relationship between a variable assignment and a subsequent use of the variable is analogous to the relationship between a rule which generates a particular consequent and a subsequent rule in the reasoning chain which uses that consequent in its antecedent. In a rule-base coverage approach to the testing of expert systems we focus on the generation of consequents (the "then" portion of a rule) during rule firings and their subsequent use in the antecedents (the "if" part) of other rules. In particular, the coverage measures, in part, examine whether each non-goal conclusion resulting from a rule firing is subsequently used in the antecedent of another rule in a reasoning chain that correctly culminates at a goal of the system.

Whereas traditional software testing techniques are often based on a "path" through the program graph, in rule-based systems we want to emphasize the entire subgraph delineated by an inference chain. Using the DAG structure described above, we define an *execution path* within the representation of a rule-based system as the sub-DAG that corresponds to all the rules fired along a particular reasoning chain executed due to a specific set of facts. We can think of a single rule firing as involving all nodes and edges in the graph that correspond to the components of the rule. Therefore each *execution path* will include the source node, the sink node, all nodes corresponding to rule consequents, all nodes corresponding to antecedent components and full antecedents, all edges linking antecedent components to operator nodes, and all edges involved in antecedent-consequent links formed by the rule connections used in the reasoning chain.

With the concept of an *execution path*, we can form a definition of *execution path coverage* for the DAG representation of the rule-base. Ideally, we would like to provide sufficient test data to cause every possible

execution path of the DAG to be traversed, which corresponds to the firing of all rules in every possible combination. However, this is usually not reasonable in the testing environment.

In TRUBAC we implement the following hierarchy of rule-base coverage measures (RBCMs) in order to guide the selection of test data and give an objective measure of how well a test suite has covered the rule-base.

1. **RBCM1** Provide test data to cause traversal of one execution path to each goal of the system.
2. **RBCM2** Provide test data to cause traversal of execution paths such that each intermediate conclusion is reached, as well as each goal. The set of test data which satisfies this coverage measure is a superset of that which satisfies RBCM1.
3. **RBCM3** There may be many execution paths that connect a fact to a goal (and there may be no execution paths for some fact-goal combinations). Provide data such that for each fact-goal combination connected by some execution path, at least one execution path which includes the combination is executed. This requires data which is a superset of the data needed by both RBCM1 and RBCM2.
4. **RBCM4** Provide data to cause traversal of all execution paths. This will be a superset of the data necessary to satisfy RBCM3.

The intent of the coverage measures is that they will provide the user with information that will lead to the acquisition or development of additional test cases, or will lead to the discovery of errors in the rule-base. However, we recognize that one of the difficulties often faced by those who test expert systems is a paucity of test data. There is an additional feature of the TRUBAC approach to rule-base testing that can help the user gain insight into the correctness of the system even in the case of too little available test data. In addition to evaluating the four coverage measures, the DAG framework used in TRUBAC allows for the automated generation of test data which would lead to traversal of any un-traversed execution paths. This synthesized data can then be shown to one or more experts in order to determine if each synthesized test case, which is a reflection of the logic embodied within a section of rule-base (corresponding to an execution path), makes sense in the context of the problem domain which the rule-base was designed to handle.

# 5 Dynamic Analysis (Validation) with TRUBAC

TRUBAC carries out dynamic analysis (or dynamic test) of the rule-based system, using actual test data (see Appendix B for a sample TRUBAC session). Dynamic methods for testing rule-based systems are applied while the system is operating on some test case(s). Typically, dynamic methods are used to compare the system's line of reasoning and conclusions with that which an expert would give in the same situation. Dynamic methods, therefore, usually correspond to testing a program by running it on data and checking the result. These methods can usually tell the user if a rule has been used or not, but the user has no way of knowing if a rule is not used because it is unnecessary in the system, or because the test cases were too limited. While traditional dynamic methods (e.g. TEIRESIAS [Dav84]) can contribute important information about the consistency and completeness of the system, they do not consider interactions among rules during possible runs of the system. Knowledge refinement systems (e.g. SEEK [PW84] and SEEK2 [GWP85]) consider particular rules in a reasoning chain, but do not consider the *absence* of rule combinations in the executed reasoning chains. It is because of the weaknesses identified in existing dynamic testing methods that a data flow analysis approach to testing rule-based systems seems warranted. In this section

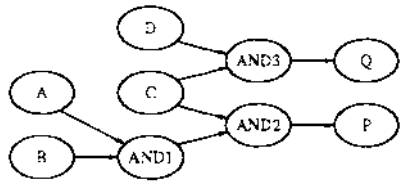



Figure 8: Graph of a complete rule as it would appear in the DAG

we discuss the application of a data flow analysis approach to rule-based systems that are represented by the DAG structure discussed above.

There are two main parts to the dynamic analysis portion of TRUBAC: percolating data through the DAG, and evaluating the rule-base coverage measures. Data must be percolated through the DAG for each test case processed by the system, and the coverage measures can be evaluated as often as is desired by the user. In order to percolate the data associated with a test case, first the input data is read and each fact node is given a value if there is one for it in the input data. Then the nodes are considered individually in a topological sort order, with an appropriate action taken based on the type of node, as follows:

- For the source, sink and fact nodes, do nothing.
- For goal nodes, OR nodes, and intermediate hypothesis nodes, if any single parent node is true then give an appropriate value to the node (could be a value or just true)
- For AND nodes, if *all* parents are true then give an appropriate value to the node.

We use both a *value* field, to indicate the effect of the **current** test case on the node, and a *dynamic mark* field, which is used to indicate whether the node had its value set during **any** test case. In addition to having a value which may be set during a test case, each node has two lists associated with it that are updated during execution of all the test cases. One is the *hold_up* list of facts which are held up on paths to the node and the other is the *pass_along* list of facts which are passed along on paths to the node. For example, consider the graph in Figure 8. Let us assume that we have a test case in which A and B are false, while C and D are true. Since A and B are both false, the result at node AND1 is false, and we say that A and B are *held up* at that node. When we consider node AND2, even though C is true the result of AND2 is false, so ultimately A, B, and C will all be on the *hold_up* list of node P. Meanwhile, since C and D are both true they will be passed along by node AND3 and will be placed on the *pass_along* list of node Q. In general, as a series of test cases is processed, some facts may be held up (in relation to a particular goal node) in one case and passed along in another case. This information is ultimately used in order to determine which fact-goal combinations have and have not been traversed by the set of test cases executed.

## 5.1 Evaluating the Rule-Base Coverage Measures

At various points during execution of the test cases, the user will want to evaluate some or all of the rule-base coverage measures. We suggest that the user try to satisfy the RBCMs in the order in which they are presented. In this section we discuss how TRUBAC determines whether or not the test cases presented actually were sufficient to satisfy the four coverage measures.

### 5.1.1 RBCM1 – Goal Coverage

For each goal in the goal list, first determine if it is in the hash table, and therefore in the graph (if not then it was in no rules and will be reported on during static analysis). If it was not marked during any test case then it is reported to the user and RBCM1 has not been satisfied.

### 5.1.2 RBCM2 – Goal and Intermediate Hypothesis Coverage

All unmarked goal and intermediate hypothesis nodes are reported to the user. If there are none then RBCM2 has been satisfied.

### 5.1.3 RBCM3 – Fact-Goal Path Coverage

For each goal on the goal list which is also in the graph, report any fact which is only on the goal's hold_up list and not also on its pass_along list. We note that a fact may be held up by one test case, in relationship to a particular goal node, and passed along by another test case. Therefore we must consider both lists. RBCM3 is satisfied if for every goal in the graph the hold_up list is a subset of the pass_along list.

### 5.1.4 RBCM4 – Execution Path Coverage

RBCM3 determines whether one execution path was traversed by the test data for each fact-goal pair for which at least one execution path exists in the DAG. However, there may be more than one way in which a fact can contribute to the conclusion of a goal. In order to find execution paths that have not been traversed, TRUBAC looks at edges in the graph. Each unmarked edge is part of some execution path that was not traversed. Therefore we start a traversal at each unmarked edge, reporting on all fact and hypothesis nodes which are part of the un-traversed execution path. We use a special marking as we do the traversal, so that a particular execution path will be reported on only once, rather than being triggered by other unmarked edges within it as they are found by the coverage check.

When the user asks TRUBAC to evaluate RBCM4, TRUBAC will offer to generate test data which would lead to traversal of the un-traversed execution paths. The user could then ask the expert(s) to review the generated data in order to determine if the synthesized test data, which is a reflection of the logic embodied within the un-traversed portions of the rule-base, makes sense in the domain the rule-base was designed to handle.

# 6 Verification with TRUBAC

As the DAG is being built, TRUBAC reports on pairs of redundant rules (same antecedent, same consequent), directly contradictory rules (same antecedent but opposite consequents), and potentially contradictory (ambiguous) rules (same antecedent, seemingly independent consequents). One of each pair of redundant rules can be left out of the final rule-base completely, and the user should correct contradictory rules or omit one of the pair. Rule pairs which are identified as being potentially contradictory should be examined by the user, but may in fact not represent an error in the rule-base.

In order to be able to provide this analysis, TRUBAC includes a hash table structure into which information is entered as the rules are processed and the graph is built. An entry is made in the table for each component of an antecedent, and for each complete antecedent. With this table, we can determine redundancy and contradiction (ambiguity) as follows. As the rules are processed, antecedent information is

entered into the look-up table. First, TRUBAC checks to see if the individual components of the antecedent are in the table. All components previously identified by the user as facts, or concluded as consequents of previously processed rules, will be in the table already. Any components not already in the table will be added. A table look-up is then carried out to see if there is an entry for the complete antecedent of the rule. The location of each antecedent in the table must be computed based on the elements found in the antecedent. If there is not already an entry for the complete antecedent, then it is entered into the table. Similarly, the consequent is looked up to determine if it is in the table already, and it is added if necessary.

If the table lookup for a complete antecedent shows that it is already in the table, and therefore already in the graph, then there is the possibility of rule redundancy or rule conflict. TRUBAC determines whether either of these situations exists by comparing the consequent of the rule currently being processed to the consequents pointed to by the antecedent node in the graph. If the consequent of the current rule is identical to any consequent pointed to by the antecedent in the graph, then there is a redundancy. If the consequent of the current rule is the negation of any consequent pointed to by the antecedent in the graph then there is a conflict. All other situations represent a potential conflict. All of these situations (redundancy, conflict, and potential conflict) are reported to the user.

### 6.1 Static Analysis

After construction of the DAG is complete, along with identification of redundant rules, simple contradictory rules and potential rule contradictions (ambiguities), static analysis of the rule-base can be carried out. The goal of static analysis is to identify cycles, dangling conditions, useless conclusions, and isolated rules. The key to the static analysis is a depth-first traversal of the DAG. The depth-first traversal provides the following information about the rule-base:

1. If there are any back-edges in the traversal, then the graph is in fact not a DAG and there are cycles in the rule-base.

2. If the traversal produces a depth-first forest, not a tree, then there are dangling conditions (an antecedent component that is not defined as a fact and is not found as the consequent of another rule).

3. If there is a rule consequent which is not known to be a goal and its node is not connected to the antecedent node of a rule which is on a path to a goal, then it is a useless conclusion.

4. If the depth first traversal generates a depth-first forest which also contains useless conclusions then there are isolated rules (rules which are not on some path from the source and also are not on some path to the sink).

In addition, the static analysis can be used to determine if there are goals of the system which appear in no rules of the rule-base.

## 7 Applications of TRUBAC

Preliminary use of TRUBAC to evaluate rule-bases indicates that it does function as desired. To date TRUBAC has been used on four rule-bases written in two different languages, for two different inference mechanisms. Three of the rule-bases were originally written for use with the EXPERT [WK84; WKKU87] system, while one was written in a generic language based on PROLOG [Pre91]. In each case the rule-base

| RULE-BASE | FACTS | GOALS | RULES | GRAPH NODES |
|---|---|---|---|---|
| Car | 14 | 8 | 17 | 44 |
| Rheum | 151 | 8 | 76 | 333 |
| Rh0184 | 864 | 28 | 940 | 2623 |

Figure 9: Size of EXPERT rule-bases and associated graphs

was translated into the form accepted by TRUBAC, and then the rule-base and various test cases were input to TRUBAC. The rule-base translation was carried out by a translator program which was customized for each of the input languages. See Appendix C for a description of the language that TRUBAC expects.

## 7.1 Results of using TRUBAC on EXPERT rule-bases

We translated 3 EXPERT rule-bases into the input language expected by TRUBAC, and then used TRUBAC to test the rule-bases. One rule-base (Car) is very small (17 rules) and suggests steps that may help when a car will not start. The other two rule-bases (Rheum and Rh0184) represent two generations of a system that diagnoses rheumatoid arthritis diseases. In Figure 9 we show the sizes of these rule-bases and the corresponding graphs created by TRUBAC. For each rule-base we had TRUBAC carry out both the static and dynamic tests. In the case of these three EXPERT rule-bases there is sample data available from when the systems were first created[1]

In the case of the Car rule-base, the graph (Figure 10) is small enough that we could get all the information provided by the static test by simply looking at the graph. The static test shows that three facts (ngas, mgas, foth) are never used in the rules. We would be able to see this clearly in the graph, as it fits on a single page. On the other hand, in a somewhat larger rule-base, such as the Rheum rule-base, the graph is larger than we can take in visually as a whole. Therefore the messages provided by TRUBAC are very useful.

While the graph is being constructed TRUBAC informs the user of rule redundancies and (potential) rule conflicts, if any exist. For example, TRUBAC reports on a potential conflict between the following two rules from the Rheum rule-base (checking for DNA antibodies):

```
if not dnap AND not dnaci then dnal(-1)
if not dnap AND not dnaci then dnam(-1)
```

TRUBAC sees that the antecedent of these rules is the same and the consequent is different. Therefore this *may* be a conflict. (Had the consequent of one rule been the exact negation of the consequent of the other rule than TRUBAC would have identified it clearly as a conflict).

In the static tests on the Rheum rule-base we are informed about a number of other problems with the rule-base. For example, TRUBAC finds that **renal** (renal involvement) is a useless conclusion. That is, it is concluded by a rule but it is not a goal and it is not used in the antecedent of any other rule. Therefore any path through the graph that ends at the **renal** node will not be able to proceed to a goal of the system. We are also told that it is rule 27 in the rule-base which leads to **renal**, allowing us to look back at the rule-base and determine if rule 27 has an error in it, should be removed from the system, or is in a part of the system

[1] Unfortunately there is no record of the systems as they were modified during development, and which test cases led to the discovery of errors.

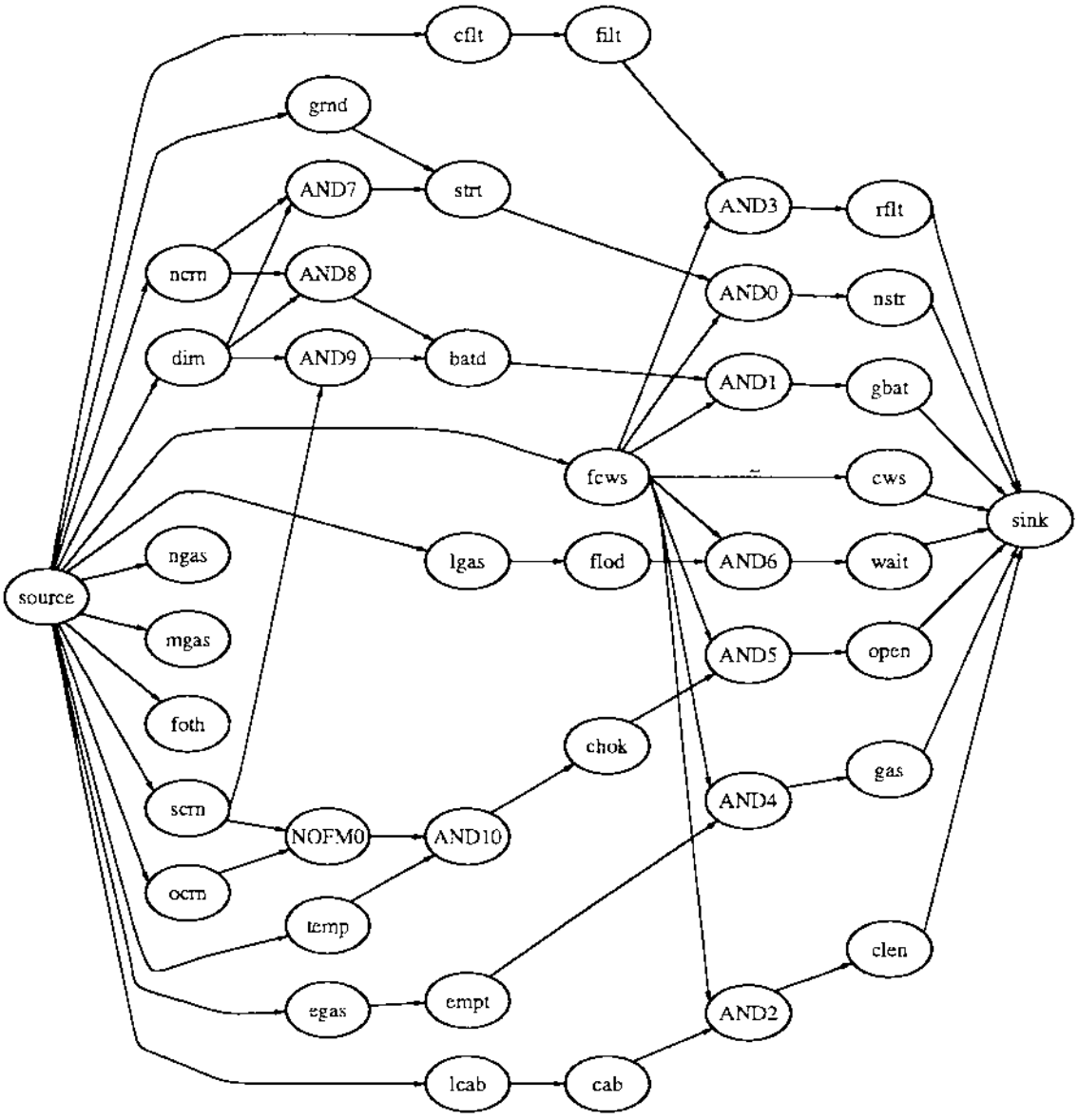



Figure 10: Complete DAG of small rule-base

| RULE-BASE | FACTS | GOALS | RULES | GRAPH NODES |
|---|---|---|---|---|
| Car | 11 | 8 | 17 | 41 |
| Rheum | 66 | 5 | 71 | 241 |
| Rh0184 | 303 | 21 | 512 | 1204 |

Figure 11: Revised size of EXPERT Rule-Bases and Graphs

that has not yet been completed. We also find out that a large number of facts are used in no rules of the system. Again this could alert us that either we plan to ask the user for more information than is necessary, or we have not implemented the entire expert system yet. This latter point would be born out by the fact that the static analysis also informs us that there are a number of goals which appear in no rules. In Figure 11 we show the size of the three EXPERT rule-bases, and the corresponding graphs, after correction of all problems identified by the static analysis.

We constructed our own test cases for dynamic test of the Car rule-base, and used existing test cases for the Rheum and Rh0184 rule-bases. Given the small size of the Car rule-base it was quite easy to construct test cases that would give complete coverage (satisfying RBCM4). In running the dynamic analysis of the Rheum and Rh0184 rule-bases we found problems with the rule-base due to the fact there were sections of the rule-base which had not yet been completed. In addition, TRUBAC exposed an error in the Rh0184 rule-base.

## 7.2 Results of using TRUBAC on Prolog rule-bases

We also tested a rule-base which was written in a generic PROLOG-based rule language [Pre91]. The intention when this language was created was that it would serve not as a language for writing expert systems, but instead as a language into which other rule-base languages could be translated. Therefore it could potentially serve well as an intermediate point between other rule-base languages and the language expected by TRUBAC.

We translated one Prolog rule-base into the input language expected by TRUBAC, and then used TRUBAC to test the rule-base. The rule-base contained 101 rules and makes suggestions of what kind of tape to use based on the environment in which the tape will be used. Translation into the TRUBAC format and subsequent graph construction results in 373 nodes, based on the 101 rules, 79 facts, and 21 goals. TRUBAC was then used to carry out both the static and dynamic tests. Unfortunately, in this case we did not have available test data from the original rule-base developer. Instead we actually used TRUBAC to guide construction of test cases, thereby mimicking the steps a rule-base tester might follow when using TRUBAC to evaluate a rule-base.

During graph construction no conflicting or redundant rules were reported. The static analysis reported 21 facts that were used in no rules. These unused facts actually are instances of new facts which were constructed during the translation process. As such they do not indicate facts that were never used in the original rule-base, but rather indicate particular values of facts that are never used in the original rule-base. That is, if TRUBAC lists subs1_lam_chip as an unused fact, then we can conclude that no rule in the original rule-base used the value lam_chip of the fact subs1.

Since we had little knowledge of the real meaning of the rule-base, we started by creating one test case for each goal, based on one rule that concludes each goal. Because of the way in which the test cases were

constructed it turned out that all 21 goals could be achieved with only 20 test cases. At that point we considered RBCM2 (all goals and intermediate hypotheses), which showed only 5 intermediate hypotheses which had never been concluded during consideration of the first 20 test cases.

An additional 4 test cases allowed us to satisfy RBCM2. We note that while these tests were constructed by studying the structure of the rule-base, we also expect them to give the correct result as well as enable us to satisfy RBCM2.

Unfortunately, the **tapes** rule-base is very flat, with few intermediate hypotheses (only 11 rules have a consequent which is not one of the 21 goals). Therefore there are numerous ways by which each rule can be concluded, and many fact-goal combinations are possible. After RBCM2 has been satisfied, RBCM3 still shows 178 unsatisfied fact-goal combinations. This means that the execution paths that include these particular 178 fact-goal combinations were not traversed during execution of the first 24 test cases. On closer examination we found that 58 of the 178 fact-goal combinations involve the facts that lead to the few intermediate hypotheses that are part of the system. It would appear that we could, in the absence of any other test data, create test cases to cover these fact-goal combinations by using the one successful test case we have so far for each goal and replacing the basic fact used to conclude the intermediate hypothesis. Alternately, it would seem that we could put all the facts that could lead to a particular intermediate hypothesis into a single test case. However, without more detailed knowledge about the problem domain we have no knowledge as to whether this would be in keeping with real situations in the problem domain or not. We also note that, in devising test cases to cover the fact-goal combinations which arise from the intermediate hypotheses, we may also cover additional fact-goal combinations.

# 8 Conclusions

In preliminary tests the approach used in TRUBAC appears to be a useful way to guide and evaluate the testing of rule-based systems. It can carry out both static and dynamic analysis using a single representation. The quality of the test data is measured based on the degree of rule-base coverage, which can be used in addition to a measure of the accuracy of the answers provided by the rule-base for the test data.

In the future we plan to experiment further with TRUBAC, applying it to more rule-bases from different problem domains for which original test data is available. In this way we can compare the rule-base problems found by TRUBAC with those found by more traditional testing methods.

# References


[Dav84] R. Davis. Interactive transfer of expertise. In B. G. Buchanan and E. H. Shortliffe, editors, *Rule-Based Expert Systems*, pages 171–205. Addison-Wesley, Reading, MA, 1984.

[FW85] P. Frankl and E. Weyuker. A data flow testing tool. In *Proceedings of IEEE Softfair II*, San Francisco, December 1985.

[GWP85] A. Ginsberg, S. Weiss, and P. Politakis. SEEK2: A generalized approach to automatic knowledge base refinement. In *Proceedings of IJCAI-85*, 1985.

[Mes91] P. Meseguer. Structural and performance metrics for rule-based expert systems. In *Proceedings of the European Workshop on the Verification and Validation of Knowledge Based Systems*, pages 165–178, Cambridge, England, July 1991.

[Pre89] A.D. Preece. Verification of rule-based expert systems in wide domains. In *Research and Development in Expert Systems VI, Proceedings of Expert Systems '89*, pages 66–77, London, September 1989. British Computer Society Specialist Group on Expert Systems.

[Pre91] A.D. Preece. A generic prolog-based rule language. Communication with the author, May 1991.

[PW84] P. Politakis and S.M. Weiss. Using empirical analysis to refine expert system knowledge bases. *Artificial Intelligence*, 22, 1984.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[WK84] S. M. Weiss and C. A. Kulikowski. *A Practical Guide to Designing Expert Systems*. Rowman & Allanheld, Totowa, NJ, 1984.

[WKKU87] S.M. Weiss, K.B. Kern, C.A. Kulikowski, and M. Uschold. A guide to the EXPERT consultation system. Technical Report CBM-TR-94, Department of Computer Science, Laboratory for Computer Science Research, Rutgers University, March 1987.

# A Rule-Based Systems

Rule-based systems have two basic parts which are of interest to us: the inference engine and the knowledge base. The knowledge base consists of the rule-base and the working memory (or database). A rule-base is a set of rules specific to the particular problem domain. For example, a rule-base system for auto repair could have rules such as

IF the engine does not start
AND the starter motor does not turn over
THEN the problem is in the electrical system

IF the engine does not start
AND the starter motor does turn over
THEN the problem is in the fuel system

Each rule has two parts: the antecedent (the IF and AND parts) and the consequent or conclusion (the THEN part). A rule is triggered if information in the database matches the antecedent of the rule. The rule "fires" and the conclusion of the rule becomes part of the working memory.

Working memory, or the database, consists of a set of facts that describe the current situation. Generally it starts out with very few facts but expands as more information is learned about the current problem based on the rules fired.

The inference engine, or rule interpreter, has two tasks. First, it examines facts in working memory and rules in the rule base, and adds new facts to the database when possible based on rule firings. Second, it determines in what order rules are scanned and fired. If the system is designed to ask the user for more information, then the inference engine does the asking via a user interface. The inference engine is also responsible for telling the user what conclusions have been reached by the system.

# B Sample TRUBAC Session

In the following script of a TRUBAC session, comments are marked with ***. The tests were carried out on the system represented by the graph in Figure 10, the DAG generated by TRUBAC for the Car rule-base.

```
                    TRUBAC
          Testing with Rule-Base Coverage

Enter filename to be translated, or q to quit: CAR

GRAPH STATISTICS:
-----------------
          44 nodes
          14 facts
           8 goals
          17 rules

Do you want to run the static tests? (Y or N)  y

*** the static tests inform us that three facts are used in no rules
*** and could be removed from the rule-base

fact mgas is used in no rules
fact ngas is used in no rules
fact foth is used in no rules

Do you wish to run the dynamic tests? (Y or N)  y

*** Each test case is stored in a file.  File name is the rule-base
*** name with the test case number appended, e.g. car1, car2, etc.
*** A typical test case file (for test case 1) is shown
*** following this TRUBAC session.

Test case number:  1

Test case 1 concludes RFLT with CF 0.8
Test case 1 concludes GBAT with CF 0.8

*** This test case causes traversal of the execution path made up
*** of edges source-fcws, source-dim, source-cflt, source-ncrn,
*** cflt-filt, filt-AND3, fcws-AND3, AND3-rflt, rflt-sink, ncrn-AND8,
*** dim-AND8, AND8-batd, batd-AND1, fcws-AND1, AND1-gbat, gbat-sink.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  m

Test case number:  2

Test case 2 concludes CLEN with CF 0.7
Test case 2 concludes NSTR with CF 0.9

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  m

Test case number:  3

Test case 3 concludes OPEN with CF 0.5
Test case 3 concludes GAS with CF 0.9

*** Now check to see if we have satisfied all goals.
```

```
Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  1

You did not provide test data to conclude cws
You did not provide test data to conclude wait
You did not satisfy RBCM 1

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  m

Test case number:  4

Test case 4 concludes WAIT with CF 0.9
Test case 4 concludes NSTR with CF 0.9

*** check all goals again

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  1

You did not provide test data to conclude cws
You did not satisfy RBCM 1

*** check all intermediate hypothesis nodes.  At this point all
*** intermediate hypothesis nodes are satisfied.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  2

You did not provide test data to conclude cws
You did not satisfy RBCM 2

*** check fact-goal combinations.  Even though all intermediate
*** hypothesis nodes have been reached, not all fact-goal combinations
*** have been used.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  3

No path executed for the GRND - NSTR fact-goal combination
No path executed for the SCRN - GBAT fact-goal combination
No path executed for the SCRN - OPEN fact-goal combination

Would you like TRUBAC to generate test data based on
un-traversed fact-goal combinations? (Y or N)  n

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  m

Test case number:  5

Test case 5 concludes OPEN with CF 0.5

*** check fact-goal combinations again.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  3

No path executed for the GRND - NSTR fact-goal combination
No path executed for the SCRN - GBAT fact-goal combination

Would you like TRUBAC to generate test data based on
un-traversed fact-goal combinations? (Y or N)  n

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  m
```

```
Test case number:  6

Test case 6 concludes GBAT with CF 0.8

*** check fact-goal combinations again.  Now only one left.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  3

No path executed for the GRND - NSTR fact-goal combination

Would you like TRUBAC to generate test data based on
un-traversed fact-goal combinations? (Y or N)  y

For fact GRND and goal NSTR use data
fact grnd
fact fcws

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  m

Test case number:  7

Test case 7 concludes CWS with CF -1.0

*** Check all goals again

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  1

You satisfied RBCM 1

*** Since all intermediate hypotheses were satisfied before, now RBCM2
*** will be satisfied.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  2

You satisfied RBCM 2

*** Still one unsatisfied fact-goal combination.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  3

No path executed for the GRND - NSTR fact-goal combination

Would you like TRUBAC to generate test data based on
un-traversed fact-goal combinations? (Y or N)  n

*** Now we check all execution paths.  Only the path involving the
*** unsatisfied fact-goal combination is left.

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  4

Un-traversed sub-DAG detected involving
Int. hypothesis or goal strt
rule number 9
fact grnd
rule number 9
rule number 17
fact fcws
goal nstr
```

<Expr> ⇒ <Disjunct> <MoreDisjuncts>
<MoreDisjuncts> ⇒ **OR** <Disjunct> <MoreDisjuncts> | $\epsilon$
<Disjunct> ⇒ <Conjunct> <MoreConjuncts>
<MoreConjuncts> ⇒ **AND** <Conjunct> <MoreConjuncts> | $\epsilon$
<Conjunct> ⇒ (<Expr>) | <IDterm> | **nofm**(numneeded, <IDterm> <IDlist>)
<IDterm> ⇒ **not** <ID> | **between**(<ID>, <CF>, <CF>) | <ID>
<IDlist> ⇒ , <IDterm> <IDlist> | $\epsilon$

IDs, facts and goals are alphanumeric strings that start with a letter.
A CF is either an integer or floating point number.

Figure 12: Grammar for antecedent expressions in TRUBAC

```
Would you like TRUBAC to generate test data based on
un-traversed sub-DAGs? (Y or N)  n

Select: 1,2,3,4 for a RBCM; (M)ore tests; (Q)uit  q
```

# C Sample Test Case

Each test case file contains the names of facts and their value (True, False, or a numeric value). For example, the file for test case 1 contains:

```
FCWS T, FOTH F, NGAS F, MGAS T, LGAS F, DIM T, CFLT T,
LCAB F, NCRN T, SCRN F, OCRN F, GRND F, EGAS F
```

Not all facts must have a value set in each test case.

# D Language Expected by TRUBAC

TRUBAC expects the rule-base to be in the following format:

*FACTS
    <list of facts, one per line>
*GOALS
    <list of goals, one per line>
*RULES
  rules are of the form "if Expression then Consequent" where the
  consequent is a single name of either a goal or an intermediate
  hypothesis, and the expression conforms to the grammar shown in
  Figure 12.

# ClassBench: A Framework for Automated Class Testing

Daniel Hoffman
Dept. of Comp. Science, Univ. of Victoria, P.O. Box 3055
Victoria, B.C., Canada V8W 3P6
dhoffman@csr.uvic.ca

## Abstract

In 1968, McIlroy proposed a software industry based on reusable components, serving roughly the same role that chips do in the hardware industry. After more than 25 years, McIlroy's vision is becoming a reality. Off-the-shelf class libraries are now available from large software producers and also from specialized component development companies.

While considerable attention has been given to techniques for developing components, little is known about testing these components. In industry and academia, the special problems of component testing have been largely ignored.

This talk will present tools and techniques for the automated testing of class libraries. The focus of our testing is collection classes—those providing sets, queues, trees, etc.—rather than on graphical user interface classes. We also focus on programmatic testing, with little human interaction; the input generation and output checking are completely under program control.

## Biographical Information

Dr. Daniel Hoffman received the B.A. degree in mathematics from the State University of New York, Binghamton, in 1974, and the M.S. and Ph.D. degrees in Computer Science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial programmer/analyst. He is currently an Associate Professor of Computer Science at the University of Victoria (British Columbia, Canada). Dr. Hoffman's research area is Software Engineering, emphasizing the industrial application of software documentation, inspection, and automated testing. He has published more than 30 journal and conference papers and has co-authored the book *Software Design, Automated Testing, and Maintenance.* He spent the 1992–93 year on sabbatical at Tandem Computers, Inc.

# ClassBench

# A Framework for Automated Class Testing

Dan Hoffman
Department of Computer Science
University of Victoria
British Columbia, Canada

1: ClassBench—A Framework for Class Testing

## Talk Overview

- ○ Class libraries
  - what are they?
  - why is class testing important?
- ○ The testgraph methodology
  - testgraph
    an abstraction of the CUT state machine
  - oracle
    CUT operations on testgraph states
  - driver
    change and query the CUT state
    heavily dependent on testgraph and oracle

2: ClassBench—A Framework for Class Testing

## Class Library Overview

- Graphical user interface (GUI) libraries
  - window and menu creation, manipulation
  - keyboard, mouse event handling
  - substantial interaction with the environment
- Collection (or container) class libraries
  - object-oriented abstract data types
  - queue, list, set, tree, sort, etc.
  - interaction entirely through calls, return values

3: ClassBench—A Framework for Class Testing

## 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

4: ClassBench—A Framework for Class Testing

## Class IntSet

```
const int MAXSIZ 10;
class IntSet {
public:
            IntSet();
        void add(int);      // duplicateExc, fullExc
        void remove(int);   // notFoundExc
        int  isMember(int);
}
```

- After `s.add(25)`: `s.isMember(25)` returns 1
- After `s.add(25)`: `s.add(25)` throws exception `duplicateExc`

5: ClassBench—A Framework for Class Testing

## IntSet Test Requirements

- Object states
  - set size: 0, 1, `MAXSIZ`
- Operations: normal case
  - `add`, `remove`, `isMember`
  - on first, last, middle elements
- Operations: exceptions
  - `duplicateExc`, `notFoundExc`, `fullExc`
- Each operation must be applied in each object state

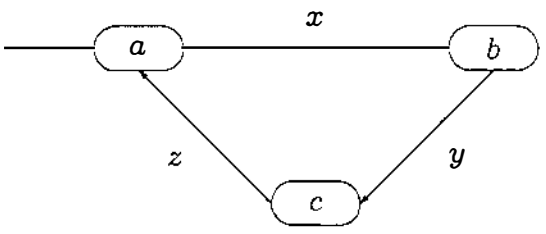
6: ClassBench—A Framework for Class Testing

## The Test Programmer's Tasks

- Design the testgraph
  - choose the states and transitions of interest
- Implement the oracle
  - CUT operations on testgraph states
- Implement the driver
  - `reset`: generate the initial state
  - `arc`: generate the transitions
  - `node`: check CUT behavior, in the current state

7: ClassBench—A Framework for Class Testing

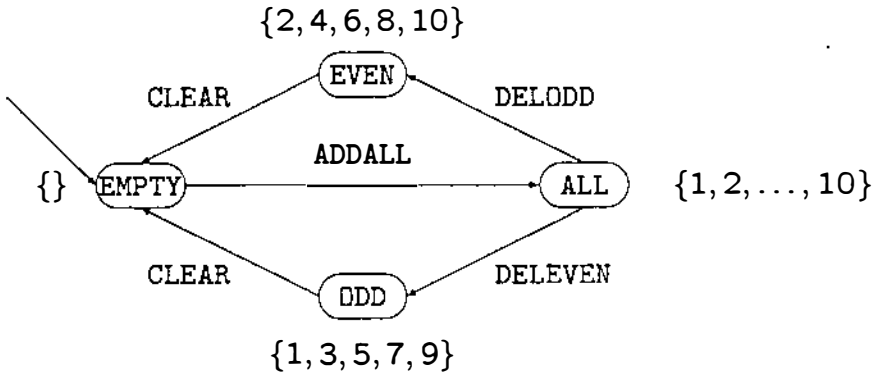
## Testgraphs




- Path, rooted path
- Coverage: node, arc, all-paths
- Arc coverage
  - NO: $\langle a, x, b, y, c \rangle$
  - YES: $\langle a, x, b, y, c, z, a \rangle$

8: ClassBench—A Framework for Class Testing

## IntSet Testgraph




- {⟨EMPTY, ADDALL, ALL, DELODD, EVEN, CLEAR, EMPTY⟩, ⟨EMPTY, ADDALL, ALL, DELEVEN, ODD, CLEAR, EMPTY⟩}



## The Test Programmer's Tasks

- Design the testgraph
  - choose the states and transitions of interest
- Implement the oracle
  - CUT operations on testgraph states
- Implement the driver
  - `reset`: generate the initial state
  - `arc`: generate the transitions
  - `node`: check CUT behavior, in the current state

## Oracle Strategy

- Test *oracle*
  - mechanism for determining output correctness
- One oracle class for each CUT
- Oracle class is like CUT, except that:
  - supports only the testgraph states and transitions
  - provides "bulk" update operations
  - does not signal exceptions
- The oracle can be *much* simpler than the CUT

11: ClassBench—A Framework for Class Testing

## Class Oracle

```
typedef enum {EMPTY, ODD, EVEN, ALL} nodeId;
class Oracle {
public:
        nodeId n;   // node identifier
        int    siz; // test suite parameter

               Oracle(nodeId,int);
        void   add(nodeId);
        void   remove(nodeId);
        int    isMember(int);
}
```

12: ClassBench—A Framework for Class Testing

## Oracle Implementation

```
int Oracle::isMember(int x)
{
        switch (n) {
        case EMPTY: return(0);
        case ODD:   return(x%2 == 1);
        case EVEN:  return(x%2 == 0);
        case ALL:   return(1); }
}

void Oracle::add(nodeId n1)
{ n = n | n1; }
```



## The Test Programmer's Tasks

- ◯ Design the testgraph
  - choose the states and transitions of interest
- ◯ Implement the oracle
  - CUT operations on testgraph states
- ◯ Implement the driver
  - `reset`: generate the initial state
  - `arc`: generate the transitions
  - `node`: check CUT behavior, in the current state

## Class Driver

```
class Driver {
public:
        void    reset(void);    // called before each path
        void    arc(int label); // called for each arc
        void    node(void);     // called for each node
private:
        Oracle  orc;
        IntSet  cut;
}
```

15: ClassBench—A Framework for Class Testing

## Utility function—loadCUT

```
// invoke cut1.add(i) for every i in orc1
void Driver::loadCUT(IntSet& cut1, Oracle& orc1)
{
        for (int i = 1; i <= orc.siz; i++)
                if (orc1.isMember(i))
                        cut1.add(i);
}
```

16: ClassBench—A Framework for Class Testing

## arc Implementation

```
void Driver::arc(int a)
{
        switch (a) {
        case ADDALL:
                orc.add(ALL);
                loadCUT(cut, orc);
                break;
        case DELODD: // ...
        case DELEVEN: // ...
        case CLEAR: // ...
        }
}
```

17: ClassBench—A Framework for Class Testing

## node Implementation

```
void Driver::node()
{       for (int i = 1; i <= orc.siz; i++)
                if (cut.isMember(i) != orc.isMember(i))
                        // report CUT failure

        if (orc.node == ALL && orc.size == MaxSize) {
                try { cut.add(0); }
                catch (fullExc) { fullExcflag = 1; }
                if (!fullExcflag) // report CUT failure
        }
        // check for duplicateExc and notFoundExc ...
}
```

18: ClassBench—A Framework for Class Testing

## The ClassBench Framework

- Test programmer supplies:
  - Testgraph
  - Test suite parameters
  - `Driver` and `Oracle` implementation

- Framework executes:

```
for each test-suite parameter value V
    for each path P returned by TestGraph
        invoke Driver::reset
        for each node/arc X in P
            X is an arc: invoke Driver::arc
            X is a node: invoke Driver::node
```

19: ClassBench—A Framework for Class Testing

## Issues: Testing the C Set++ Library

- Scaling up
  - Many classes each with many member functions
  - ⇒ Identify and exploit similarities among classes
  - ⇒ Develop prototype suites for simplified classes

- Operations of pairs of objects

- Iterator nondeterminism

- Template classes

20: ClassBench—A Framework for Class Testing

## Scaling Up

- Set classes
  - Set, Key Set, Key Sorted Set, Sorted Set, Bag, Key Bag, Key Sorted Bag, Sorted Bag, Map, Sorted Map, Relation, Sorted Relation, Heap
- Sequence classes
  - Deque, Equality Sequence, Priority Queue, Queue, Stack, Sequence
- Tree classes
  - N-ary Tree

21: ClassBench—A Framework for Class Testing

## Issues: Testing the C Set++ Library

- Scaling up
- Operations of pairs of objects
  - Many operators, e.g., union, combine two objects
  - ⇒ Develop techniques to easily test combinations of CUT states
- Iterator nondeterminism
- Template classes

22: ClassBench—A Framework for Class Testing

## Operations of Pairs of Objects

```
void Driver::checkOpPlus() const
{
   for (int n = EMPTY; n <= ALL; n++) {
      // create and initialize lhs CUT
      Oracle lhsOrc(n,orc.maxSize());
      TestCut lhsCut(Oracle(n,orc.maxSize()));

      // compute and check result
      lhsOrc += orc;
      CHECKVAL(&(lhsCut += cut),&lhsCut); // return value
      CHECKVALBOOLEAN(lhsCut == lhsOrc,1); // object state
} }
```

23: ClassBench—A Framework for Class Testing

## Issues: Testing the C Set++ Library

- ◯ Scaling up
- ◯ Operations of pairs of objects
- ◯ Iterator nondeterminism
  - Iterator retrieval order is nondeterministic

  ⇒ Use range/count/sum checks
- ◯ Template classes

24: ClassBench—A Framework for Class Testing

## Iterator Nondeterminism

```
void Driver::checkIter() const
{       int element,sum = 0,count = 0;
        TestCutIter iter(cut);

        while (!iter.isEnd()) {
                element = iter.next();
                CHECKVALBOOLEAN(orc.isMember(element),1);
                count++;
                sum += element; }
        CHECKVAL(orc.size(),count);
        CHECKVAL(orc.sum(),sum);
}
```

25: ClassBench—A Framework for Class Testing

## Issues: Testing the C Set++ Library

- Scaling up
- Operations of pairs of objects
- Iterator nondeterminism
- Template classes
  - Unclear which element types to choose for testing
  - ⇒ Parameterize test suites by element type
  - ⇒ Test one builtin type and one user-defined type

26: ClassBench—A Framework for Class Testing

## Template Classes

```
class E : public String {
public:
        E() { }

        E(int x) // the key: user-defined type conversion
        {
                char s[80];
                sprintf(s,"%d",x);
                (String)(*this).String::operator=(s);
        }
};
```

27: ClassBench—A Framework for Class Testing

## Results: Testing The C Set++ Library

| | Prototype | | | C Set++ | | |
|---|---|---|---|---|---|---|
| | Set | Sequence | Tree | Set | Sequence | Tree |
| TestGraph nodes | 4 | 4 | 4 | 4 | 4 | 4 |
| TestGraph arcs | 5 | 5 | 5 | 5 | 5 | 5 |
| `Oracle.C` (LOC) | 93 | 108 | 92 | 123 | 109 | 184 |
| `Driver.C` (LOC) | 225 | 261 | 328 | 566 | 786 | 997 |

28: ClassBench—A Framework for Class Testing

## Conclusions

- The importance of class testing
  - reuse depends on reliability
  - reliability depends on thorough, repeatable testing
- Effective testing of collection classes
  - will exploit the call-based interface
  - *not* keystroke/mouse-click capture/playback
- The testgraph methodology
  - testgraph: abstraction of the CUT state machine
  - oracle: provide CUT operations on testgraph states
  - driver: `nod/arc` calls triggered by testgraph traversal

29: ClassBench—A Framework for Class Testing

# Jump-Start Your Stalled Improvement Effort

Hilly Alexander, Margie Davis

ADP Dealer Services
2525 SW First Avenue
Portland, Oregon 97201-4760

**Key words:** process for change, implementing change, process documentation, management commitment, formal inspections

**Audience:** anybody who introduces new techniques or methods into an organization

## Abstract

A process change must be introduced at three levels in the organization. Each group has a different set of needs. You must satisfy all three groups to gain acceptance of the new process. In this paper we describe our experience in implementing the formal inspection process. We demonstrate how we learned to understand and meet the differing needs of our developers, middle managers, and senior managers.

## Biographies

During this effort, Hilly and Margie were members of the Software Engineering Process Group, supporting software development process improvements.

Hilly Alexander has worked at ADP Dealer Services for 8 years as a quality assurance analyst, project leader, and supervisor. Before joining ADP she worked as a marketing researcher, technical librarian, customer support technician, and programmer. Sharing information about processes and techniques that work and helping people adopt them is her top priority.

Margie Davis has worked at ADP Dealer Services for 10 years. Margie's experiences as a junior high school teacher, Army statistician, programmer, analyst, project leader, hospice volunteer, and Big Sister have helped her appreciate the challenges of change. Her favorite accomplishments are helping people turn their brilliant ideas into reality.

# 1 Introduction

When we embarked on our process improvement effort, we knew we had to understand how to introduce change into our organization. We knew we had to be change agents, find a senior management sponsor, and find developers who were willing to try something new. One aspect of the change process surprised us: the importance of selling change at three levels in the organization. We knew we had to provide a cost justification to get buy-in from senior managers and provide developers with an easy-to-use process. We did not anticipate that middle managers have different needs. In this paper we describe our experience introducing formal inspections. We discuss where our implementation stalled and what we did to jump-start the effort.

We thought the implementation of formal inspections would be easy. Formal inspections satisfy four criteria we think are essential for success:

- ☑ proven process
- ☑ management commitment
- ☑ documentation & training readily available
- ☑ defined metrics

We will focus on what surprised us and why it takes so long to implement change. We found that the roll-out process involves more than providing training and documentation to the users. We revised our process for change to reflect this insight.

# 2 Who we Are

ADP Dealer Services is the third largest division within Automatic Data Processing, Inc., an independent computing services company. Dealer Services provides computing solutions to automotive dealerships in North America and Europe. The division employs more than 3000 associates, including over 350 associates in software development in Portland, Oregon. Development teams include project leads, analysts, programmers, writers, trainers, and quality assurance specialists.

As members of the Software Engineering Process Group (SEPG) we were responsible for facilitating process improvement efforts in the software development organization. The SEPG coordinates action planning activities, organizes the implementation of new methods and technology, maintains the process documentation and data, coaches and trains, and communicates process improvement goals and activities to associates. Refer to "Managing the Software Process" [Humphrey89] and the "Software Engineering Process Group Guide" [TR24] for detailed descriptions of the SEPG's role.

# 3 Change Process

Organizational change has been described from different perspectives, for example, the **change leader**'s perspective (how to plan and implement) or the **end users**' perspective (how people adopt change). The models agree on these major points:

- there are stages of organizational change
- it is important to select the right people at each stage.

The change leader's perspective focuses on the importance of planning, communicating, implementing, and fine tuning. The change process requires planning just like any other project. The typical phases of an adoption and implementation life cycle are:

- needs assessment
- product or process selection
- piloting
- fine tuning
- release
- maintenance and support [TR31]

The end users' perspective highlights how different people react to change. In order to make the change effort successful it is important to understand these differences. Some people are eager to be on the leading edge while others prefer to wait for proven processes. People do not adopt even the best idea or technique overnight. They may try it once or twice but then need encouragement and support to make it part of their normal process.

It is important to identify and recruit the right people at each step of the change process. Priscilla Fowler describes adopter populations (as defined by Everett Rogers [Rogers83]) as follows:

- "Innovators" love new ideas and do not need documentation,
- "Early Adopters" are eager to try something new but need some guidance,
- "Early Majority" are willing to try new methods for business reasons,
- "Late Majority" want predictability and low cost, and
- "Laggards" are the last to change [Fowler93].

A "late majority" adopter who expects predictability and complete documentation will be less comfortable during prototype or pilot. "Innovators" on the other hand may not have the patience for validating process documentation. A person may fall into different populations depending on the change. A person may be an innovator if you are changing a manual process but in the early majority if you are automating a manual process.

The following table highlights Fowler and Levine's change life cycle phases, how the Software Engineering Process Group (SEPG) supports each phase, and who the best supporters are based on how they adopt change.

| Change implementation life cycle | SEPG support for each phase | Best Supporter |
|---|---|---|
| **Needs assessment** | Identify a process need<br>• complaints from developers;<br>• management directive | innovator |
| **Product or process selection** | Find suitable technique and define usage<br>• conference or training seminar;<br>• magazine or book;<br>• learn from other companies, new associate;<br>• prototype the process | early adopter |
| **Piloting** | Initiate pilot<br>• draft process document;<br>• test training and documentation for usability;<br>• collect and report metrics | early majority |
| **Fine tuning** | Expand pilot<br>• finalize process documentation;<br>• finalize training | early and late majority |
| **Release** | Roll out to the organization<br>• news articles in local newsgroup;<br>• memos to management for circulation in their teams;<br>• distribute process documentation | late majority |
| **Maintenance and support** | Encourage and monitor use<br>• track planned and actual use;<br>• provide training;<br>• update process documentation | late majority |

Understanding the change process and its effect on people helps us recognize the delays and resistance to change that commonly occur. We know that implementing a new method or technique takes time but are surprised by how long some stages take.

# 4 Our Process for Change

Our process for change is based on the change implementation life cycle listed in the previous section. We combined or expanded phases based on our experience within our culture.

## Process for Change

| | |
|---|---|
| Define<br>• Identify need<br>• Draft or acquire process or standard | We combine these tasks to re-enforce interdependence. If we jump to solutions (a specific process or standard), this reminds us to make sure we understand the need. If we get caught up in complaints about a need, this reminds us to look around for how others solved a similar problem. |
| Test<br>• Prototype for applicability<br>• Formalize for repeatability<br>• Quantify success criteria for measurability<br>• Pilot test for usability | We stage our testing and fine tuning of a process or standard into several test steps. This allows us to find the best type of supporter for each purpose and to include many teams in the testing. We find that wider involvement means wider ownership in the final result. Most teams end up with their own process expert by the time the testing is complete. |
| Institutionalize<br>• Train practitioners to use<br>• Provide visibility of use<br>• Train managers to plan and track | We expand the release phase to specifically address the needs of our three audiences. For us, release implied simply making a process or standard available. We have learned that making a new process or standard routine (institutionalized) requires much more effort. |
| Support<br>• Provide on-going training and support<br>• Fine tune the process from lessons learned | We have not yet reached this stage for formal inspections. We realize that no process succeeds unattended for very long. It requires updating to keep in line with the growth of the overall development process. |

The following sections describe each process step and provide guidelines and examples from our experience.

## 4.1 Define

### Identify Need

Identify a problem area within the organization and select a suitable improvement effort.

**Guidelines**

- Select or develop a process improvement framework
- Get direction from a Process Improvement Steering Committee
- Observe project team problems

Refer to a process improvement framework to identify efforts that have a reasonable chance for success. The Software Engineering Institute's Capability Maturity Model, for example, identifies the key process areas for improvement based on an organization's maturity level. Trying to introduce an advanced method into an immature organization has a high risk of failure. We would not attempt to implement causal defect analysis in a company that barely manages to record, track, and correct defects.

Form a Process Improvement Steering Committee to direct the improvement activities. Steering Committee members should be managers who represent the groups that will be affected by the improvement effort. They provide the management perspective on whether the effort answers a business need and fits within the overall strategic goals of the company. Find managers who will support change in their teams.

Listen to the process complaints and pains in the product development teams. Look for common causes for those complaints. The greater the pain, the more receptive people are to trying a different technique. The improvement effort should address the need felt in the teams and map into the improvement framework.

### Draft or Acquire Process or Standard

Research existing proven processes.

**Guidelines**

- Attend conferences, review literature
- Network with other companies
- Investigate your own company best practices

Attend conferences or seminars to learn about and practice the process. Review magazines and books for proven ideas. Many process problems have been addressed by others. You can save time by learning from the industry leaders.

Talk to peers at other companies who use the process. Learn from their experience. You can gain valuable insights from other people's efforts to try something new.

Find out if anybody in your company is already using this or a similar process and use their knowledge and experience as a starting point. When you find a good process in-house, you begin with a strong support base.

### Our Experience

Our senior vice president gave us our first improvement effort. A consultant recommended that we use formal inspections as a cost-effective improvement to our review process. The formal inspection process offers a streamlined alternative to the walkthroughs which many teams consider tedious and time consuming. To respond quickly to the need, we next needed to find formal inspection training and documentation.

Because the formal inspection process is well documented, we hired a consultant to provide in-house training. The training included a management overview session and a practitioner course. The training materials included process documentation. We advertised the class to managers by identifying who should attend and encouraged them to send team members. We hoped to save time by using the inspection process from the training materials. To determine whether the process could be implemented in our organization and culture, we next needed to try it in a real project.

## 4.2 Test

### Prototype for Applicability

Recruit associates who are willing to try the new process in their project and evaluate its applicability.

**Guidelines**

- Involve limited, low risk projects
- Provide direction
- Use the process yourself

Find people who attended process training and are eager to try the new process on their projects. Prototype participants should be comfortable with using draft documentation. They should work on projects that are small, or have low priority. The project schedule should allow the participants time to learn the new process. The Process Improvement Steering Committee can help identify potential participants and projects.

The process improvement group should direct the prototype. Development team members who have to meet deadlines rarely can devote the necessary time to experiment with the new process. The improvement group focuses on the process implementation issues while developers use the new technique in their projects.

Use the process. It is difficult to identify snags and find ways to remove them if you only have theoretical knowledge of the process. The improvement group participates in the prototype, learning from experience as well as feedback.

### Formalize for Repeatability

Identify the documentation and training package that will enable the teams to use the process consistently.

**Guidelines**

- Form a work group
- Provide simple documentation
- Offer short, tailored training

Form a work group of people who will use and improve the new process in their projects. It is critical to find members who:

- can champion the technique in their team
- will share their experience with the work group
- want to develop or improve process documentation.

Process documentation should be simple. People will resist adopting a process that appears complicated. The documentation should include:

- a process flow - this high-level view shows the process steps
- process step narratives - short paragraphs describe the objectives and participants at each step
- checklists - these lists identify the tasks and responsibilities to help process users plan, prepare, and execute the process steps
- forms - these help the users organize planning information and collect metrics data.

Process training should be short and tailored to the audience. An inspector, for example, needs to learn about the formal inspection process and his/her role, but does not need to learn how to moderate an inspection. Moderators need to learn about all roles and understand the entire process.

### Quantify Success Criteria for Measurability

Define the metrics that will allow us to measure process performance.

**Guidelines**

- ➢ Collect efficiency and effectiveness metrics
- ➢ Compare results to other companies

Define the measures to collect from the process. Make the data collection as easy as possible. Ensure that the measurement is part of something larger, e.g. an organizational focus on improvement. Use the Goals - Questions - Measures model to tie measures to decision-making. For example:

- Goal: Reduce the cost of rework
- Questions: How long does it take on average to correct a defect found in an inspection? How does this compare to correcting a defect found in system test?
- Measures: Number of defects reworked that were found in inspections; number of hours spent on rework; number of defects reworked that were found during system test; number of hours spent correcting those defects.

Find data from other companies as a benchmark for comparison. Robert Grady's "Practical Software Metrics for Project Management and Process Improvement" [Grady92] is a good source for data.

### Pilot Test for Usability

Identify obstacles to ease of use.

**Guidelines**

- Find champions
- Test and learn

Find people who believe the new process presents an improvement over their current practice and will champion its introduction in their team. Look for people who will follow the documented process and report process problems.

Hold regular meetings to debrief the champions and maintain momentum. Participate in the process. Listen to problems, identify systemic issues and correct the process documentation as necessary.

### Our Experience

We prototyped the formal inspection process in six teams on a variety of work products: planning documents, requirements and design specifications, and code. The SEPG supported the teams by providing training, participating in inspections, and bringing people together to share their experience. Participants evaluated their experience and identified what worked and what did not work. The teams reported what they learned and recorded metrics from the prototype inspections. They concluded that the process was useful but documentation and training materials needed to be simpler. To practice the process efficiently, we next needed to revise the formal inspection materials.

The first set of documentation we acquired was complete but cumbersome to use. Process users had to reference several sections in the document to gather all the information needed to complete a task. Who wants to browse through a manual just to prepare for and conduct an inspection?

We continually simplified the documentation and training materials to give people just what they needed to know. The work group provided feedback on the key elements to document and train. Much of this information is now presented in checklists or forms to serve as a reminder at exactly the time it is needed. Appendix A includes copies of the presentation slides. Slides 4, 5 and 6 show samples of our inspection process documentation. Our team training was reduced to 30 minutes.

As we distributed the process to more people, we wanted to be sure it was consistently and effectively practiced. We next needed to quantify how the process was working.

We used the data collected from all formal inspections the teams had done. We presented the normalized data and our findings to the Process Improvement Steering Committee:

- defect detection rate (number of defects found per hour),
- rework rate (number of defects fixed per hour),
- logging rate (number of defects recorded per minute in the Defect Logging Meeting), and
- defect density (number of defects per unit of size).

We included metrics reported by other companies for comparison. The data from our own design and code inspections provides a benchmark for moderators to monitor the efficiency of their inspections. Appendix A, presentation slide 8 shows two of our efficiency metrics.

Because our numbers were "in the ball park" with other companies who successfully use inspections, our Steering Committee published a recommendation to use formal inspections for design specifications and code. At the same time they asked us to initiate collection and reporting of comparable metrics from other forms of peer reviews and from testing activities. This additional data would allow us to evaluate the effectiveness of the new process. To tell if the process could be adopted by other developers, we next needed to test it in more teams.

Our project teams have practiced peer reviews, i.e. walkthroughs and desk checking, for years. This meant that pilot teams were able to internalize the inspection process and started to benefit after only two or three experiences. Although the process documentation was sound, other issues arose.

The pilot teams were very concerned about the confidentiality of the metrics data. One team assigned its most concerned member to keep the metrics data safe. We now have a guideline describing which data to present when managers ask about the effectiveness of the process in their team. Our senior vice president issued a policy statement directing managers not to use defect data in performance appraisals. With these acceptance issues identified and addressed, it was time to turn the inspection process over to the development teams.

## 4.3 Institutionalize

### Train Practitioners to Use

Assist trained team members in training their peers.

**Guidelines**

- Teach only what they need to know
- Conduct just in time training session

Process training should be tailored to the intended audience. Process users need only enough information to exercise their part in the process. The training should include:

- an overview of the process
- guidelines on when the process is most effective
- a handout which includes the process steps, checklists and samples of the forms for reference.

Provide process training when it is needed. Training is most effective when it immediately precedes the first use of the new technique. The training materials should be:

- portable so training can be presented almost anywhere
- easy to use
- include answers to the most common process questions.

### Provide Visibility of Use

Determine how to feed information about the process back into the organization.

**Guidelines**

- Regularly publish which teams participate
- Report planned and actual practice

Data on process usage should be regularly published. This provides feedback to the teams at the company-level. The teams' practice of the new process is quantified and visible to the company.

Find a way to capture the teams' plans for using the process. Merely collecting data about actual use leaves out an important indicator to implementation obstacles. If project teams are not planning to use the process, you need to ask why and address the issues. If actual use is significantly less than planned use, you need to identify the reasons and address them.

### Train Managers to Plan and Track

Provide training to middle managers.

**Guidelines**

- Identify project management and middle management needs
- Provide training on planning and tracking

Determine what project and middle managers need to support the adoption of the process on their project or in their team. When managers do not participate in the process and have no prior experience with it, they have difficulty understanding its benefits and its impact on performance and productivity. They also are not sure how to support the practitioners. This is especially true when the new process is different from the way they have always done things.

Provide interactive training on planning and tracking the new process. This training is most effective if it can draw on the managers' experience with a similar process. The training should include:

- planning and tracking data for a similar process collected from the managers
- planning and tracking data that reflects the company's experience with the new process, for example, average effort and elapsed time
- industry numbers for comparison
- guidelines for best application of the process.

### Our Experience

Our trained moderators now present the 30-minute inspector training module to new inspectors as part of the inspection kickoff meeting. Making the training part of the process removed the difficulty of scheduling training classes. Managers find the brief training attractive because they can easily fit it into the project schedule. It is also easy to re-train inspectors if they have not used the process for some time and need a refresher course.

We now use a consultant to provide a two-day moderator training class. Moderators participate in an in-house certification program following the formal training.

As the SEPG stepped back to let the process prove its worth, we saw a drop in inspection activity. To figure out why, we asked questions to understand who was using inspections and why, and who was not using them and why.

Our first surprise was that senior managers thought their teams were using inspections simply because they had told them to. Now we publish a table that lists the number of formal inspections, walkthroughs, and desk checking activities performed each month in each department. We distribute the report to senior managers and trained moderators. The monthly report prompts senior managers to encourage process usage if their teams practice the process less than other teams. Appendix A, slide 9 shows our monthly activity report.

Senior managers wanted to see who was planning and completing peer reviews. To increase visibility of process usage at the project level we require project managers to report planned and actual peer reviews. A new section in the monthly project reports to senior management captures that data. This also offers the SEPG the opportunity to review this data and contact teams that need more support or encouragement to use inspections. We were surprised at all the reasons we heard why teams were not using formal inspections: do not know about the process, do not know when to use them, do not report to the chief moderator when they use them. The SEPG can offer help in overcoming these obstacles.

The lack of buy-in and support from middle managers was our biggest surprise. We realized that we had overlooked a critical audience. We had intentionally kept managers from participating in the inspection process to insure metrics confidentiality of individual inspections. We overlooked the critical role they play in making any process happen within the project. Without the personal experience with the new process, managers found it difficult to plan for the practice. Since the data from individual inspections is confidential, managers did not know how to measure the effectiveness of the practice. We developed guidelines for sharing data from formal inspections with managers. We provide management reporting only at the summary level and only when that summary information can not be directly tied to an individual's performance. Our management training module and individual consulting with managers uses the managers' experience with walkthroughs to help them set expectations and plan for formal inspections. Appendix A, slide 10 shows topics in our manager training.

We continue to linger in the "Institutionalize" step today.

## 4.4 Support

### Provide On-Going Training and Support

Support the released process.

**Guidelines**

- Coordinate and train
- Collect and document feedback from users

Provide whatever assistance the teams need to implement the process. Support includes:

- finding experienced process users to assist inexperienced teams
- training team members and managers
- using the process yourself to maintain expertise

- helping managers implement the process
- explaining again and again and again the benefits and best use of the process.

Interview teams about their experience with the process. Publish their success stories and lessons learned.

### Fine Tune Process From Lessons Learned

Review the process.

**Guidelines**
- ➢ Analyze the implementation
- ➢ Improve the process

Review the process periodically to verify that it still meets the needs of the organization. Examine each step in the process and ask whether it needs adjustment. Improve the process.

### Our Experience

This is the step we are entering. We promote inspections by maintaining management visibility and encouraging management commitment. Teams with a self-proclaimed champion of formal inspections were the first to adopt them. Teams whose managers' bonus plan includes the requirement to perform formal inspections are using the process to keep their managers happy.

We underestimated the effort and attention needed to roll out formal inspections. We better defined the institutionalization step to insure practitioner buy-in, project management buy-in, and compliance measurement. Our customized formal inspection process itself initially did not require any changes. We attribute this primarily to the thorough testing in different teams, and the fact that the process users were new to formal inspections. Our process was the only one they knew. We offer suggestions through newsgroups and observations. More recently some of the process documentation has been questioned by new associates who have used formal inspections at other companies and are new to our variation of formal inspections. Finding ways to keep process use consistent where it counts while allowing some flexibility is our current challenge.

## 5 Summary

It was easy to find industry data to prove the cost benefit of formal inspections to senior management. We collected those same metrics during our pilots for comparison. This provided "intellectual" support and prompted senior management to state that they expected teams to use inspections. Unfortunately, in our organization it is not enough if the "captain" says "make it so!" New processes do not get institutionalized here without continued senior management focus. Once you have buy-in, continuous visibility of process use is a primary need of senior management.

Selling inspections to practitioners was easiest. They want more effective methods. Our challenge was to come up with simple process descriptions supported by checklists, forms, and tools. Many people still only want to learn as much about the process as they need to know to do their part. Training works best here when it is done as part of using the process for the first time and only lasts 30-60 minutes. Once a strong

process has been developed, continuous re-enforcement of process steps is a primary need of the practitioners.

Getting buy-in from middle managers was the toughest job. We finally understood why. They have become successful doing things the "old" way. We were not giving them help in how to plan and track the effectiveness of formal inspections. This was a critical lesson: training managers requires a much different approach than training practitioners in how to use the process. Once you know how to plan and track, the continuous support in individual project planning is a primary need of middle managers.

To adopt a new process in the organization you must meet four pre-requisites. Here is what we did to satisfy them for formal inspections (text in italics):

- ☑ process defined and documented - *work instructions: process description*
- ☑ standards and procedures exist - *work instructions: forms, checklists*
- ☑ all practitioners trained - *moderator training, process training, management training*
- ☑ management commitment - *monthly project reports, bonus plans*

Institutionalization is an iterative process. You must continually focus on these activities. Here is what we are doing (text in italics):

- ☑ institutionalize the process bottom-up - *development work groups, test work groups, one-on-one*
- ☑ have all practitioners participate in the process - *team training*
- ☑ enforce ethics and confidentiality - *confidential metrics*
- ☑ measure project compliance periodically and consistently - *monthly activity table*
- ☑ have a strategy, committed and communicated - *Steering Committee recommendation to use inspections, monthly state-of-the-practice report*
- ☑ drive improvement toward increasing productivity, reducing cost - *monthly report metrics, lessons learned*

Our next step is to determine whether there is a correlation between product release quality and the use of formal inspections.

## References


[Fowler93] Priscilla Fowler. Software Technology Transition Tutorial at *SEPG National Meeting April 27, 1993*

[Grady92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement* Prentice Hall, Englewood Cliffs, 1992.

[Humphrey89] Watts S. Humphrey. *Managing the Software Process* Addison-Wesley Publishing Company, Inc., 1989.

[Rogers83] Everett Rogers. *Diffusion of Innovation*
The Free Press, New York, 1983, p. 247.

[TR24] Priscilla Fowler and Stan Rifkin.
*Software Engineering Process Group Guide*
Technical Report CMU/SEI-90-TR-24, ESD-90-TR-225, Software Engineering Institute, September 1990.

[TR31] Priscilla Fowler and Linda Levine.
*A Conceptual Framework for Software Technology Transition*
Technical Report CMU/SEI-93-TR-31, ESC-TR-93-317, Software Engineering Institute, December 1993.

## Appendix A

Presentation slides (starting on the next page)

# Jump-Start Your Stalled Improvement Effort

Hilly Alexander
Margie Davis

1



## Jump-Start Your Stalled Improvement Effort

Process we should be using:

Seems to be stalled because:

| | |
|---|---|
| __ need not understood | __ insufficient control |
| __ process is inefficient | __ no mgmnt commitment |
| __ bad documentation | __ immaturity of use |
| __ insufficient training | __ other?_____ |

2

# Jump-Start Your Stalled Improvement Effort

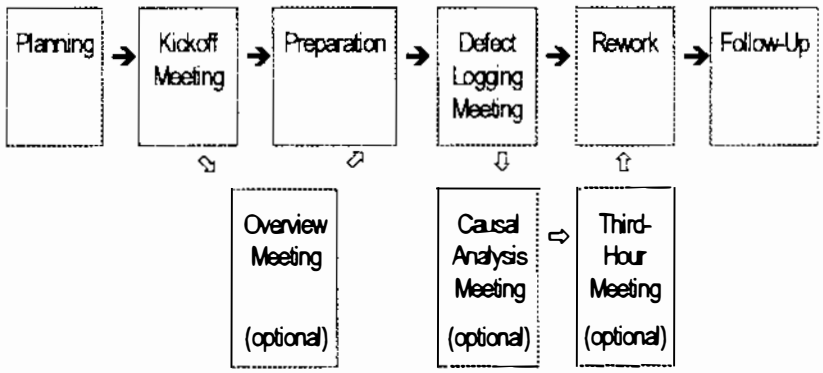
## Process for Change

| | |
|---|---|
| Define | Identify need<br>Draft or acquire process or standard |
| Test | Prototype for applicability<br>Formalize for repeatability<br>Quantify success criteria for measurability<br>Pilot test for usability |
| Institutionalize | Train practitioners to use<br>Provide visibility of use<br>Train managers to plan and track |
| Support | Provide on-going training and support<br>Fine tune process from lessons learned |

3



## Process Flows




4

# Jump-Start Your Stalled Improvement Effort

## Process Step Descriptions

Preparation

Participants: Author, inspectors, scribe

Objective: During preparation the inspectors find and record defects. The author is an inspector. Inspectors record defects on the work product or on the Inspection Defect List. Defects are recorded in seven words or less. This will improve the efficiency of the defect logging meeting. The scribe prepares copies of the Inspection Defect List for the defect logging meeting.

5 

## Process Checklists

| Inspection Step | Actions |
|---|---|
| Planning | ☐ Check entry criteria<br>Work product labeled for reference<br>Parent document is available<br>☐ Choose inspection team<br>☐ Schedule meetings<br>☐ Prepare *Inspection Announcement*<br>☐ Make copies for kickoff meeting<br>☐ Record planning time on *Inspection Summary Report* |

6 

# Jump-Start Your Stalled Improvement Effort

## Lessons Learned

Getting Started

- Not everyone has the same comfort-level giving and receiving direct feedback on defects. Moderators need to be especially sensitive to that in inspections.

Roles

- When you moderate in another group, spend time with the author to make sure your expectations on how the process will work are the same.

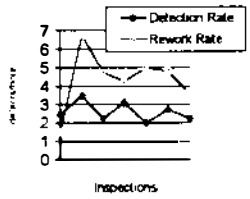
Meetings

- The scribe needs to sit mid-table to be able to hear everyone clearly.

7



## Management Recommendation




Design Specifications: Sample size = 7

Detection Rate is the number of defects found per hour spent in planning, individual review, and the logging meeting. Hewlett-Packard reports an average detection rate of 2.5 defects per hour. Our average Detection Rate for design inspections is 2.7 defects per hour.

Rework Rate is the number of defects fixed per hour by the author. Rework includes researching and making changes. Our average design Rework Rate is 4.4 defects per hour.

8

# Jump-Start Your Stalled Improvement Effort

## Management Visibility

| Inspections Completed | Jul 94 | Aug 94 | Sep 94 | ... | Jun 95 | FY 95 to date |
|---|---|---|---|---|---|---|
| Team 1 | | | 1 | | | 1 |
| Team 2 | 2 | | 1 | | | 3 |
| Team 3 | | | | | | 0 |
| Team 4 | | | | | | 0 |
| ... | | | | | | |
| SEPG | | | 1 | | | 1 |
| TOTAL | 2 | 0 | 3 | | | 5 |

9



## Project Manager Training

Choosing work products and review methods

- ▼What does your team review now?
- ▼Risk assessment
- ▼Review methods in project situations

Scheduling reviews into projects

- ▼How much time do your reviews take?
- ▼Inspection metrics (for estimating)

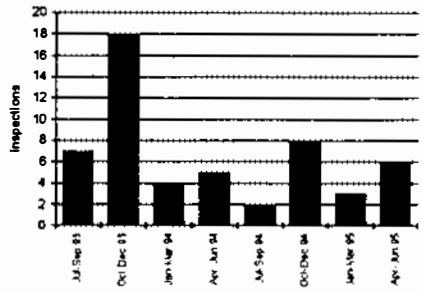
Tracking review effectiveness

- ▼How do you know if your reviews are effective?
- ▼Evaluating metrics

10

# Jump-Start Your Stalled Improvement Effort

## Inspection Activity




Jul - Nov 1993:
Pilot Test

Dec 1993:
Practitioner Training

Mar 1994:
Management Visibility

May - Jul 1994:
Management Training

Dec 1994:
Project Visibility

11

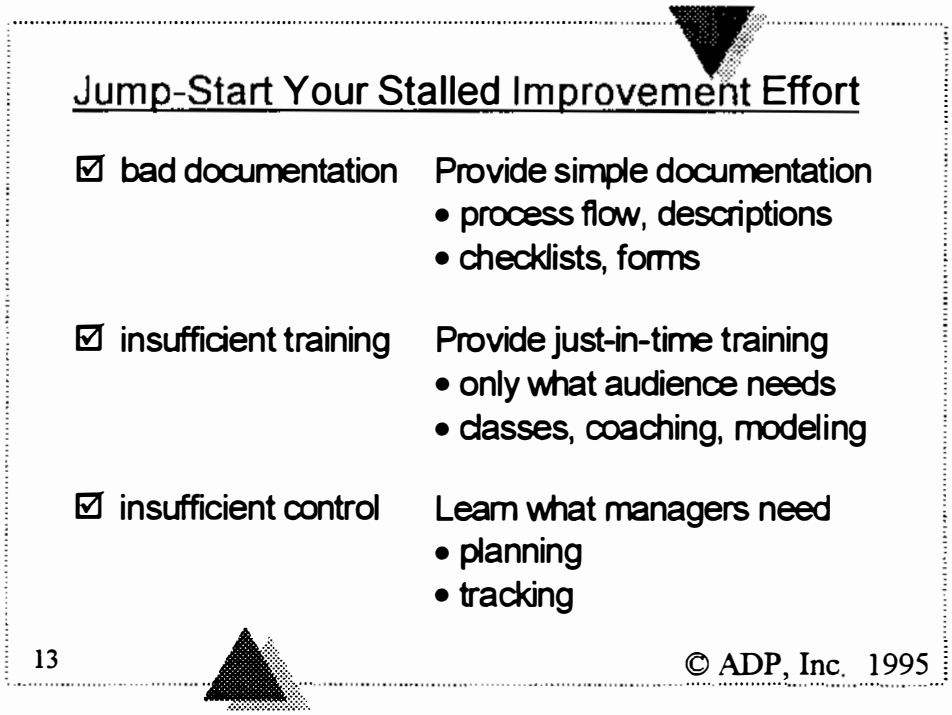


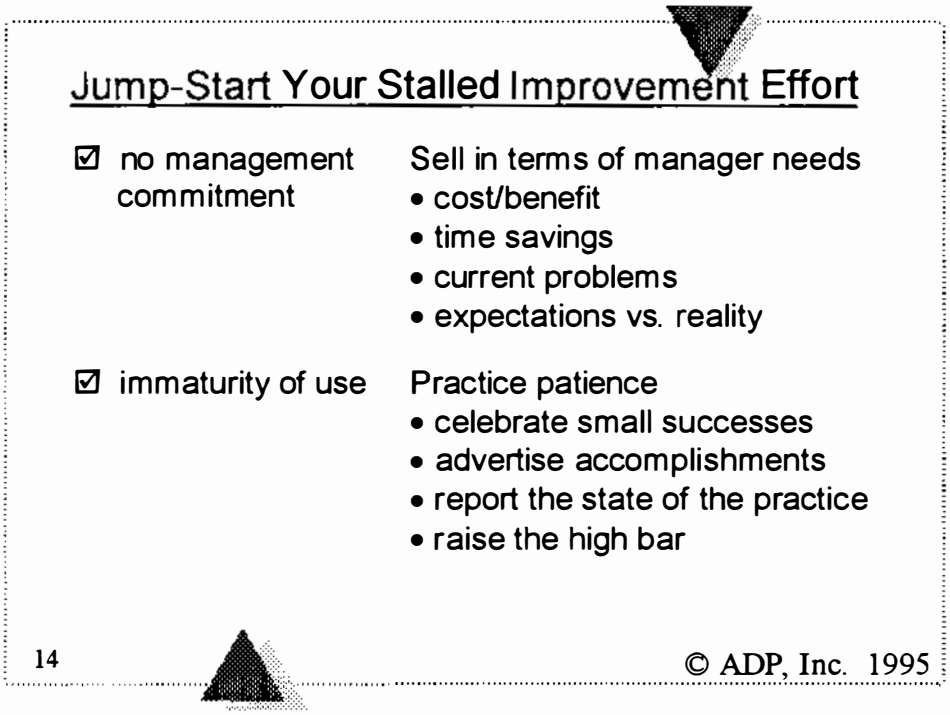
## Jump-Start Your Stalled Improvement Effort

| | |
|---|---|
| ☑ need not understood | Do more research, evangelize<br>• literature search<br>• other companies<br>• best practices in your company |
| ☑ process is inefficient | Test and improve the process<br>• applicability<br>• repeatability<br>• measurability<br>• usability |

12

# Jump-Start Your Stalled Improvement Effort



## One Company's Struggle to Produce Quality Software

This paper outlines how a small company took steps toward establishing quality assurance procedures in their development process. The software life cycle contains many pitfalls along the coding and testing process that can prevent small companies from creating solid applications. During the design and coding phases of the development process, quality assurance practices such as coding standards and a software methodology can make the testing phase run more smoothly. Defects will turn up quickly if the programming team commits to following documentation and design procedures early in the product life cycle. Effective defect tracking and proper testing techniques will reduce the testing time needed to find and fix defects, thus allowing a small company to avoid maintenance releases and move on to diversifying its product base.

By implementing quality assurance procedures early on, one small company has been able to enjoy profitable relationships with two major OEM customers. Their tape backup software is consistantly on the top seller list for OS/2 applications.

**Anton Webber**
**Quality Assurance Manager, MSR Development**
**4619 North Street**
**Nacogdoches, Texas, 75961**
**(409) 564-1862**
**fax (409) 560-5868**
**75213.3162@Compuserve.com**

**Michael M. Pickard**
**Dept. of Computer Science**
**Stephen F. Austin State University**
**Nacogdoches, Texas 75962**
**(409) 468-2508**
**f_pickardmm@titan.sfasu.edu**

**Anton L. Webber works at MSR Development as Software Quality Assurance Manager. He received a B.S. degree in computer science from Stephen F. Austin State University in 1994. He has also worked as an independent contract programmer.**

**Michael M. Pickard, currently assistant professor of computer science at Stephen F. Austin State University, has more than twenty years of experience in software development as a programmer, system analyst, project leader, and consultant. He received the Ph.D. in computer science from Mississippi State University in 1993. He received the M.S. in computer science and B.A. in mathematics from Mississippi State University in 1971 and 1969 respectively.**

## Introduction

Bad software practices have been the death knell of many fledgling companies struggling to develop commercial software. In an effort to produce software that is competitive in the market, many small companies will rush a product out the door, only to find themselves over-burdened with troubled calls from unhappy or bewildered customers. One company's efforts to establish quality assurance programs have led to successful software development and company growth. By setting up these practices early on, the company has positioned itself to become a leader in the OS/2 software market.

## Company History

MSR Development began as a three man team of young developers who saw a small market niche in the tape backup industry. Having come to know one another while pursuing computer science degrees at Stephen F. Austin State University, the company founders looked for a market area that had not become saturated. At the time (1992), IBM's new operating system, OS/2, had yet to establish itself prominently enough to catch the attention of larger software development companies. The developers began to lay plans for a tape backup product for OS/2 environments.

Small attempts were made initially to come up with a formal design of the product, but for the most part, system development did not follow any particular methodology. Each of the young men upgraded their personal computers to prepare for the development effort they were about to undertake. A fourth computer was used for a bulletin board system (BBS) that would be needed to distribute test builds to beta testers. Although the initial product was delayed due to IBM's release of OS/2 2.0 and 2.1, the entrepreneurs grew more excited each day. Their enthusiasm continued to climb as activity on their BBS showed major interest in their product. Each developer was committed to putting in twelve to sixteen hour days in order to deliver the product before their savings ran out.

The emphasis in the development of the first release was on getting the product completed and released for sale. Perfunctory testing was done, but in the end 250 beta testers were needed to expose the defects that did not turn up in testing. Only 10% of the beta testers were fully committed to providing the much-needed information to the developers. After fourteen months, MSR delivered an OS/2 tape backup solution for floppy tape drives. This type of tape drive connected to the floppy controller interface making it inexpensive to build. Almost immediately after shipping, the phones started ringing. The need for a technical support staff became apparent in order for the engineers to continue their work on new releases. In order to head off the impending disaster, MSR made steps to organize the development process and introduce quality assurance practices.

## Recognition of Need for Quality Assurance (QA)

As might be expected, neither the large beta program nor the six month delay of the deliverables exposed all the defects in the first release. During the six month period between October of 1993 and April of 1994, MSR shipped its first release and two maintenance releases. All three shipped with numerous defects that only became apparent when the product hit the market. Many bad habits formed over the next few months. For example, no reported defects went through any tracking procedure. Instead, a note was made concerning each defect and was informally passed to a developer, who would fix the problem and discard the defect report.

The need for formalized testing and quality assurance procedures quickly became evident and was emphasized during the following summer. MSR needed to support the popular high speed adapters that many potential customers were using to double their backup speed. Another beta program was launched that included over one hundred beta testers. The tech support department became overwhelmed with defect reports. The extensive beta period that followed pushed the company into near bankruptcy. What had looked to be a promising future a few months before was becoming a nightmare. The quality assurance department was started in order to analyze the development effort and provide a team of testers that could review a product thoroughly before it was shipped.

**Coding Standards**

The first step most software teams make toward creating a stable and productive development environment is to establish a coding standard. The first time a developer has to look through the code of one of his peers, he immediately notices the subtle differences in coding styles that make the code seem unreadable. The addition of one indention or the placement of a single bracket will cloud the meaning of a statement, thus causing minor confusion. "We need a standard!!", he'll say. MSR Development wrote their coding standards in order to establish a writing style upon which all three programmers could agree. Each one felt that the other two wrote "sloppy" code. This was the primary motivation behind creating the standard, but it is by no means the greatest benefit derived from it. These benefits were only realized after the coding standard was put in place.

The most obvious advantage is code reusability. Reusing code not only saves development time, but saves in the time needed to properly test a given set of modules. This also ensures higher quality code. As long as unit level testing was performed on the code originally, stringent tests can be avoided in future implementations, thus saving time.

Another important benefit of coding standards is the documentation requirements. Depending on the level of testing that QA is willing to commit to, documentation can play a key role in designing and implementing white box testing routines. A tester armed with a good set of documentation and a basic knowledge of internal data types can write extremely effective test code.

Establishing the coding standard marked an important step in the growth of this small company. It was the first commitment toward formalizing their development process.

**Defect Tracking**

It is amazing to think that MSR distributed three releases of their software before any attempt was made to start a defect tracking system. Although they realized that defect tracking was necessary, several months passed before they were able to implement it. The results were readily apparent. During those initial months, MSR was receiving an average of two technical support calls per shipped product! This large influx of information required good software to manage it.

Good defect tracking systems will allow multiple users to access the database. This can be accomplished in two ways. First, some packages are affordable enough to install on each of the developer's machines. A database is located on the network server and can be accessed easily at any time. The advantage of this type of setup is that the developer can perform a variety of queries in an attempt to find similarities between different defects. Track Record by Underware is a good example of this type of application. Second, a good package like DCS by Intersolv can interface with popular e-mail applications such as CC-Mail and Microsoft Mail. This allows one person, the bug master, to control the database and distribute information easily through e-mail. DCS can also be installed on several machines and configured for each type of user. Testers, developers, and management all have different access rights to the database. MSR began using Track Record but eventually moved to DCS for long term reasons. The DCS package is more suited for larger companies.

A very important aspect of defect tracking is the reporting procedure. MSR practices two types of procedures: active and passive. Active reporting is the most common type. It is usually used for reporting obvious errors in the application such as memory leaks or misspelled menu items. This type of defect will occur on every machine that runs the applications. A report is sent to the bug master who logs it and notifies the appropriate developer. Passive reporting is used for the more obscure defects such as questionable hardware compatibility or defects that are difficult to isolate. MSR's technical support department distributes a Frequency of Error (FOE) report that shows how many users are experiencing certain categories of problems. The bug master can log each defect and go to work collecting more information on the defects with high frequency rates. This reporting method is effective during times of busy activity in technical support. The data is automatically collected by a customer support software package called Clientele by Answerset Corporation. Clientele is a 'call' tracking software used to keep up with a wide variety of information concerning users who call in for technical support or sales information.

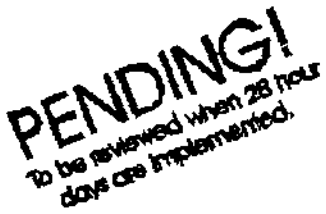
The FOE report becomes especially handy when the defect involves device drivers. Certain hardware configurations can be tracked down by collecting more information on all users who fall into a particular error category. Passive reporting will also eliminate the single defect occurrences that may or may not have been user error. If one of these types of errors shows up in a defect report, a developer might invest too much time trying to fix a defect that only occurs during a thunder storm at high noon. Nothing will affect revenue like a wild goose chase. Patrick O. Conley, president of Abraxas Software, Inc. talks about this dilemma of software support by saying, "My experience with [software developers] is that when they first come out with a great product, it sells well. They pay for their advertising, and everything is great. Then about six months after introduction, they start spending all their time doing support. They never get back to upgrading the product and the revenue drops. Now they're trapped. They have no time to develop new products. They get squeezed and die" (CONL89).

**Software Methodology**

The software methodology defines the processes that must be followed in order to push a product through its software life cycle. Below is a basic software life cycle as presented in (LEDG87).

**Project Definition**
**Formal Specifications**
**Design**
**Construction**
**Testing**
**Installation**

Adoption of a well-defined software methodology was the first comprehensive attempt to make order of the chaos within the company. It was becoming apparent that MSR was very lucky to have made a success of a product whose architecture was quite literally thrown together. It also became obvious that the company would not be able to do it again. This undertaking was the first time many of us had ever heard of the Capability and Maturity Model (CMM) as defined by the Software Engineering Institute (SEI). The development staff had several meetings to discuss a method for defining, implementing, and testing a product. Each day some mention of the CMM came up, and all would agree that such a program would greatly improve the company's ability to develop quality software. As we looked over the requirements, we became increasingly aware that our methodology paled in comparison to the formalized guidelines of such a system. The large stack of papers that cluttered each meeting would be quickly reshuffled so that those guidelines would wind up on the bottom. A large stamp was placed on the guidelines stating:




Management realized the importance of those guidelines but was satisfied that we were making a small step in the right direction. A commitment has been made to revisit this issue when the company increases the size of its development and quality assurance staff.

The methodology addressed several ideas that would eventually play an important role in quality assurance and testing. The testers now have the ability to start designing test code early in the product life cycle. Development has to establish formal specifications for every module and every function before any coding could be done. No changes can be made to those specifications unless everyone involved on the project is notified of the change through a formal set of procedures. While this may seem like a lot of red tape, this procedure and many others bridges the communication gap between the two groups.

The methodology requires the project team to create data flow diagrams and structure charts. These design documents can be used as tools for the testers in their efforts to design the best possible test code. Testers should have intimate knowledge of how the application works so they can expose weak spots in its implementation. Those in charge of testing should refuse to check code that they know nothing about. The QA department can start designing test code before any coding is started by the developers. This greatly reduces testing time, thus reducing time to market. As soon as a particular module is finished, each of its functions can be compiled into the test code and quickly analyzed for possible defects. If any defects are found, the developer can be notified while the module is still fresh on his or her mind.

Following the software methodology is an important part of the development process. Without it, the application could end up as a large kludge of functions without any basis for being considered quality software. The most immediate return on following a methodology concerns time-to-market. With a significant reduction in development time, a healthy investment in testing can be made without affecting the release date. MSR has seen every indication that an increase in design documentation is accompanied by a very substantial decrease in development and testing time.

**Software Testing Procedures**

Once we had successfully tracked down and fixed the defects that had extended our beta period, the first major revision of the product was released to market. MSR managed to attract the attention of a multi-billion dollar company that wanted an OEM version of the software. It seemed that their current OS/2 solution was also contracted out to a competitor of MSR Development. Their product defect counts were so high that the larger company was forced to search for a better product to provide its customers. Our first encounter with formalized test procedures came at an awkward moment. We had just finalized our contract when their QA department called requesting our test plans. Naturally, they wanted to get a head start so they would be prepared for our upcoming release. Up until that point, all of our testing was strictly a loose set of black box tests. Nothing formal had ever been put on paper, so we set out to develop MSR's first set of test plans.

**Test Plans**

The test plan was designed as a check list of tests to be performed in a specific order. System information is recorded at the beginning along with vital information about the test procedure. The figure below describes a set of tests that are classified in two categories: those expected to work and those expected to fail. It is very important to test the error handling ability of the software. Here is a small excerpt of a black box test plan. This gives a basic idea of the types of tests that need to be performed:

| | **Normal Installation** | **Results** |
|---|---|---|
| | Perform an installation without using the mouse | |
| | Perform an installation over the previous install | |

| | **Abnormal Installation** | **Results** |
|---|---|---|
| | Install to an invalid drive (non-existent) | |
| | Install to a drive with insufficient disk space (RAM drive) | |
| | Cancel an install before completion | |
| | Install to a directory name greater than 8.3 on a FAT system | |

Figure 1. Excerpt of a test plan.

Creating the test plan can be done early in the development process. There is no need to wait until any coding is finished before a test plan can be drawn up. As long as the specifications are properly documented and adhered to, much of the work can be started early in the project life cycle. Test plans are an excellent tool for beta testers. Many times a product goes out to beta without a manual to accompany it. To a good tester, a robust test plan can give a clearer

picture of the product's capabilities than a manual. Running a comprehensive test plan can give the tester the experience needed to attack the product with more skill at finding defects.

**Automated Testing**

In an effort to enhance our QA department, MSR obtained permission to tour the software research and development department of their largest OEM customer. It was here that we got ideas for improving some aspects of the development and testing process, such as automated testing and project builds.

Automated testing runs the test application on all machines connected to a network. This procedure is designed to provide the developers with vital debugging information and system performance statistics. A loader program is installed on each target machine early in the coding phase. This program is designed to download any test application from the network, run it, and upload debugging information when it is through. At night, every company employee can load the test program, turn off their monitor, and go home. A complete system backup is performed before the user returns for the next day. The information from the test is quickly compiled into a report and is distributed to team members each morning. A fill list of encountered errors and complete debugging information shows the developers which machines exhibit common problems. The following is a list of the tasks performed during an automated test run:

1. Check for the user information file. If none exists, collect the following information.
    a. Ask the user to enter system information.
        (tape drive type, controller card, etc...).
    b. Ask the user what time they will return the next day.
    c. Save this information to a file.
2. Logoff the network.
3. Logon to the test account. (This allows the user to access the test directories.)
4. Download the latest test build.
5. Download the test script.
6. Logoff.
7. Execute the test script.
8. Log back on to the network
9. Upload the log file.
10. Perform a system backup.

It is necessary to plan for the automated test capability in the application early in the design phase. A pseudo-test can be set up if the application can be run with a full set of command line arguments and has extensive logging capabilities. This situation does not provide adequate unit-level testing but it will still provide excellent information if hardware compatibility is an issue.

One problem encountered when initially installing the test program on all the machines was an overwhelming reluctance to run the test applications. There is a general feeling that test code can contaminate one's machine. Everyone outside of development and testing will have some excuse as to why their machines are too vital to the company to be running questionable software. The automated test program must be trustworthy before anyone will participate in the program. The possibility of data loss can only be a problem if the project is not implemented properly. If the business team is unwilling to run the product in-house, then selling it to the public is out of the question. Before MSR's tape backup software was released, every developer had to perform a full backup, format his hard drive, and restore the data.

**Unit Level Testing**

All PC enthusiasts have had to deal with a system crash at some point. Those who deal with this enough have learned how to find the problem quickly: start pulling components out until the problem is isolated. There is no way to deal with the system as a whole. One must test every functional unit. And so it is with software testing. This is the most vital element in any testing procedure. "...all will come to nothing and the project will fail, if the unit-level software, the individual routines, have not been properly tested. Quality Assurance that ignores unit-level

testing issues is a construct built on a foundation of sand." (BEIZ90) When you see the number of defects found during this phase, it is almost easy to consider black box testing a waste of effort. The testing team should invest at least 80 percent of its time on unit level testing. If the product is fully functional, code-coverage tools demonstrate which modules need to be tested more. In most cases, a surprisingly large amount of code never gets executed. MSR uses this type of testing on its device drivers. According to MSR's defect list, 46 percent of all defects found originated from the device driver. A calling routine has been written to ask the driver to perform small sub tasks such as tape drive initialization, reading tape segments, and formatting a tape. Since the driver runs at the kernel level of the operating system, debugging information is difficult to get to. By testing these sub tasks individually, MSR is able to isolate problems more effectively.

**The Build Process**

During the coding for the first product shipped, each developer would complete a particular module and copy the necessary files to a build directory where the compile would be performed. No revision control system (RCS) prevented them from breaking something they had already fixed. After the first release was made, an RCS was finally purchased and the shipping product was checked in. Even then, all builds were performed on the last files to be completed. It was up to the author to test the code before doing a check-in. One developer was designated as the official builder. He was responsible for backing up the RCS directory and storing the tape off location to prevent loss of data due to some disaster.

When MSR's development process began to expand to multiple projects, it became necessary to set up a structured build environment. The first step was to dedicate a machine for builds. Since the development staff is so small, all of the PCs are connected to a network with a single server. The compiler is installed on each machine. For a larger development staff, it is easier to simply purchase a site license and install the compiler on the central development server. The compiler can be periodically copied down to each development machine to maintain identical compilers on all machines.

The main benefit of performing builds on a dedicated machine is that QA has full access to the source code. All unit level testing needed for a particular module can be done just before the code is incorporated into the build process. When the project leader notifies QA of an update to a module, QA will test it and update the check-out routine to incorporate the change. The team members can check in as many modules as they need into the revision control system, but no code can be used in the build unless it meets documentation and testing qualifications.

**Debug Code**

MSR strongly recommends that each new employee in the development group complete a reading list before they tackle any projects. Several of those books suggest that incorporating "debug code" into functions is a good practice for preventing defects from consuming too much personal testing time. Writing Solid Code by Steve Maguire (MAGU93) and No Bugs! by David Thielen (THIE92) suggest using preprocessor directives (#ifdefs) and assertions when implementing debug code that would prove useful in tracking down the source of defects. The developers at MSR quickly adopted the idea and met with QA to establish a more thorough definition for using debug code. We came up with a small set of debug levels that could be used depending on the level of information needed during a debugging process.

Here is a brief summary of Maguire's and Theilen's use of debug code. A #define is used to turn on the debugging code. #ifdefs surround all the debug code, thus allowing the preprocessor directives to create the debug code. An example of the resulting code appears in Figure 2.

```
#include <stdio.h>
#define DEBUG

BOOL ClrBuffer( far *pBuf)
{
#ifdef DEBUG
```

```
        printf(" Entering ClrBuffer function");
#endif
        .
        .
        CODE
        .
        .
}
```

Figure 2. Example of the use of debug code.

The debug code needs to provide adequate information without bogging down the program. Debug levels were established to provide varying levels of information according to the developers needs. We came up with the following debugging levels.

1. Debug level one - The entry and exit of each function are recorded to a log file.
2. Debug level two - All parameters passed into the function are logged.
3. Debug level three - This is simply a checkpoint within the function.

By activating a debug level, you also activate the previous level. For example, a designation of debug level two will implement level one and level two code, but not level three code.

Our first try at using debug code ended up being scrapped because the developers felt that a full implementation of the concept made the functions unreadable. The coding standard only required a developer to use at least level two, thus leaving level three to be implemented at the developer's discretion. Here is an example of our original implementation.

```
#include <stdio.h>
#include <error.h> /* defines the return code types such as 'returncode'.  */
#define DEBUG
#define DEBUG2

returncode ClrBuffer( far *pBuf)
{
#ifdef DEBUG
        printf("\n Entering ClrBuffer function");
#endif
#ifdef DEBUG2
        printf("\n Argument :%x", pBuf");
#endif
        .
        .
        CODE
        .
        .
#ifdef DEBUG
        printf("\n Return Value %d", errorcode );
#endif
        ret errorcode;
}
```

Figure 3. First implementation of level one and level two debug code

It is necessary to flush the write buffer to insure the integrity of the data being gathered. Under OS/2, printfs may not get a chance to display their information before the application halts execution. It may become necessary to send the information to a log file.

The other option we explored involved heavy use of macros. Macros work better that functions in this case because there is no jump to the location of code when the debug routine is called. The code is simply copied to the desired location at compile time. Examples of debug macros would be Debug_Open(), Debug_Close(), Debug_Assert(x,i,y). A debug level zero would result in a blank macro. The following listing shows how figure DC2 looks under the current implementation

```
#include <stdio.h>
#include <error.h> /* defines the return code types such as 'returncode'. */
#define use_debug_open
#define use_debug_arg
#define use_debug_ret
#define use_debug_close

returncode ClrBuffer( far *pBuf)
{
        debug_open();
        debug_arg("%d",pBuf);
        .
        CODE
        .
        .
        debug_ret(errorcode);
        debug_close();
        ret errorcode;
}
```

Figure 4. Macro implementation of level one and level two debug code.

Figure 5 shows a section of the macro listing that MSR has implemented.

```
#ifdef use_debug_assert
 #define debug_assert( b ) DebugMgrAssert( (b), FUNCNAME, __LINE__, __FILE__ )
#else
 #define debug_assert( b )
#endif
```

Figure 5. Excerpt of the header file containing the debug macros.

Whenever two processes, or threads, are running simultaneously, the log file can become cluttered with information. We briefly entertained the idea of having multiple defect logs but decided that a single log file provides more information by showing the sequence of events that occur before a defect rears its ugly head.

MSR does not track the defects found by the developers during their personal test runs. Therefore, it is impossible to show exactly how much benefit comes from implementing debug code. In the future, we hope to design debugging routines directly into test code in such a way as to allow the tester to specify the debugging level at run time. This will provide useful information from field testers during beta test cycles. It would be very helpful to log a defect along with information that shows in which function the application failed. In OS/2, a common error received during a test run could be an operating system halt. Execution of the program will halt and the tester can check the log file and give useful information as to what occurred before the error was generated.

**Project Audits**

One small procedure that MSR established was a project auditing guideline which was designed to make sure that each team leader was following the methodology. The whole point of committing to the methodology was so that an overlap of work could occur between development and testing. It is easy for a development team to start coding before much of the documentation is finished. This may allow coding to be started early but the testing group will suffer in the end. A simple auditing procedure can prevent this from happening. Many times the development team will forget to document something as simple as the function hierarchy chart. Everyone on the team has a clear picture of the program's architecture, so nobody ever thinks to write it down. The audit should simply be a reminder to the team leader that other teams may

find the documentation useful. The following outline shows how MSR designated when audits should be performed within the company.

**Project Requirements Phase**
QA Prep meeting with the Development Team
All submitted requirements must be available for review in a database.
A variable schedule must be created to weigh requirements v. development time.
The final schedule determines the exact requirements to be included into the project.
*1st Project Review (Audit)*

**Definition Phase**
Data Flow Diagrams
Structure Chart
Functional Specifications
*2nd Project Review (Audit)*

**Coding Phase**
Coding and Testing
*3rd Project Review (Audit)*
- How were component tests performed?
- Were functions written according to specifications?

The audit is performed by the QA Manager and turned in to the Development Manager for review.

**Project Teams**

In his book, The Mythical Man Month, Brooks defines the makeup of a good development team (BROO75). He calls this the "surgical team". A brief summary of the team members is listed below:

| | |
|---|---|
| **surgeon** | The chief programmer and team leader. |
| **copilot** | Alter ego of the surgeon. Provides the surgeon with helpful solutions. |
| **administrator** | Interfaces with the administrative machinery of the whole organization. |
| **editor** | Fine tunes the technical documents produced by the surgeon. |
| **secretaries** | Assists the administrator and editor in their responsibilities. |
| **program clerk** | Maintains the technical records of the team. |
| **tool smith** | Ensures the adequacy of development tools needed by the team. |
| **tester** | Designs and implements system test cases. |
| **language lawyer** | Expert in the intricacies of the programming language. |

For an organization as small as MSR, this team concept provides only a framework for establishing the development team. In order to implement the "surgical team", our developers would have to wear many hats. Figure 6 shows the typical layout for a team at MSR.

**Team Leader**
1. surgeon
2. editor
3. tool smith

| **Team Member** | **Team Member** | **Team Secretary (Auditor)** |
|---|---|---|
| 1. copilot | 1. Graphical User Interface expert | 1. program clerk |
| 2. system expert | 2. language lawyer | 2. tester |
| | | 3. QA advisor |

Figure 6. Typical programming team for a small company.

One person from QA is assigned to the project. This person attends all team meetings and acts as the team secretary. As secretary, this person will help handle the documentation effort and other administrative duties, and is the one member who will not produce any code that will find its way into the final product. The secretary also acts as the tester. This relationship with the team allows the tester to have full access to and a working knowledge of the project

documentation and architectural design. During the design phase, this secretary can help influence the team as they attempt to come up with a suitable implementation plan that will lend itself easily to the testing effort. This relationship also helps solidify the communication gap between the developers and the testers in QA. Animosity between these two groups is natural due to their inherent roles as "makers" and "breakers". By allowing a tester to become part of the team, the two groups can focus on the common goal of producing quality software.

Another advantage to the "team secretary" arrangement is that it provides a defined promotional path within development. Being a team secretary and member of the QA and testing group can be a good stepping stone for inexperienced developers. Neophyte programmers can see first hand the importance of writing solid code. Team leaders can see how well this team member works in the development environment by his or her ability to design and implement test code independently.

**Watson Meetings**

At the end of the 1994 year, MSR Development decided that it was necessary to enhance the technical support department. The popularity of the product had resulted in a flood of technical support calls. The support group provided its customers with many avenues for receiving technical support and sales information. Besides the common telephone access, MSR maintains a BBS that provides access to the latest demos and a place for leaving messages for a support technician. Compuserve members (or Internet users) can leave messages in the OS/2 B vendors forum. Customers can always send faxes that are answered as quickly as possible. Supporting all of these services forces the support staff to handle a large amount of information. A tightly monitored communications flow had to be established in order to properly track all of the incoming defect reports. The defect reporting method was enhanced to prevent reports from being lost in the confusion. A mechanism for providing the support staff with crucial information was created. We call it a "Watson Meeting".

MSR frequently deals with defects that seem to appear out of thin air without a clue of where they originated. Many weeks could go by before anyone in support will learn enough about the defect to start providing the unhappy customer with meaningful support. If QA and development are having difficulty pinpointing the cause of the problem, a meeting will be called to discuss the defect and share information. Sherlock Holmes would often discuss the intricacies of a case with his companion, Watson. This verbal exchange always helped the detective solve the case. Hence the name, "Watson Meeting".

The Watson Meeting can be called by anyone involved in handling the defect. The usual attendants would be: the technical support staff, the QA person assigned to hunt the defect, and the developer most closely involved with "the case". The meeting usually starts out with an architectural overview of the application. Discussion is then focused on possible causes and available work-arounds. The meeting is mostly a brain storming session. Even if no resolution can be made immediately, the technical support staff is now equipped with an arsenal of knowledge concerning the nature of the defect.

**Software Release Checklist**

As the quality assurance department began to establish itself within the company, other departments looked to QA to perform various other roles. The QA group is considered an effective tool for "putting out fires" that pop up unexpectedly. Up to this point, QA tracked defects, performed software testing, and managed the beta program for new releases. Developers began asking QA to test builds for individual modifications to modules in the code. At times, technical support would not be able to help a customer with their problems. QA would be asked to "take over the case." Upper management sometimes got calls from large PC manufacturers complaining that our software did not work with their hardware but would on their competitor's. Believe it or not, a very large sale was pending on two thousand notebook computers because our software acted strangely on this hardware. It worked fine on another brand that the buyer had considered using. QA spent many hours attempting to identify the problem. The cause was isolated to a problem in the hardware, which the computer vendor resolved by making changes in the manufacturing process.

Taking on extra tasks is a necessary part of business within a small company. In an effort to reduce the load on QA, management defined a strict set of guidelines for initiating testing during the coding and testing phases of the product life cycle. These guidelines are referred to as the Software Release Checklist (SRC), which is now being implemented.

The SRC defines several stages of the product life cycle. The product must pass specific criteria before it can "mature" to the next stage. The stages are as follows:

1. **Personal Build.** This is an untested version for engineering use only.
2. **Engineering Build.** This build undergoes a predetermined test set to verify a fix.
3. **Green Build.** Once coding is complete, limited tests are performed in this stage.
4. **Beta Build.** Full testing is performed before this build can be released to beta.
6. **Golden Master.** Once the beta test is complete, preparations are made for release.
7. **Released Product.**

The product is considered a "candidate" until it has passed the requirements for that stage. Most of the stages have demotion criteria that force a build back into its previous stage. For example, if too many defects are found during the beta cycle, the product is demoted back to an engineering build for further refinement. In each stage, a series of questions is asked to determine what steps are involved in promoting the build. If those steps are completed without error, the build moves up in the maturity cycle.

**Summary: Results and Recommendations**

The most critical step in implementing a quality assurance program is to make sure management is committed to the effort. Improving the quality program takes a strong commitment by all departments in the business. Sanders and Curran state, "Quality must be accepted by everyone as the company's central value. Otherwise the technical program is of no use." (SAND94) The goals of MSR's QA department are:

1. Minimize time-to-market.
2. Zero defects shipped.

In order to achieve these goals, MSR will continue to strengthen its QA program and even strive toward attaining ISO 9000 certification.

For young companies looking for ways to implement a QA program, the ideas expressed in this paper offer a solid foundation for starting effective quality practices. These ideas could have been more effective had they been implemented in a different order. The list that follows shows the steps the company should have taken in its first development effort:

**1. Establish a software methodology for designing and implementing an application.** The plan should outline every step of the development process from collecting requirements to releasing the product to market. All aspects of testing should be considered in all phases of the development process.

**2. Write the coding standards.** Attention should be given to documentation requirements, accepted writing styles, and debug code. These standards should not inhibit the programmer but aid the team in writing clear and readable code.

**3. Put together the business team for the application.** Application requirements should be collected in order to write the product definition. Marketing and distribution should also be considered by this team.

**4. Determine the composition of the development team.** The makeup of the team should fit the company's needs. Duties and responsibilities should be assigned to each member. Choose a team secretary to help design and code testing routines.

**5. Design the application.** An architectural design shows exactly how the development team will meet the needs of the product definition.

**6. Establish a testing team.** Set up the build process and check-in procedure for finished modules.

**7. Set up a revision control system.** Project management applications help control the code revision process by tracking which code revision pertains to certain projects.

**8. Set up the defect tracking system.** Defects can be tracked as early as the coding phase. If unit level testing turns up a problem, the proper communication channels should already be established.

**9. Begin coding.** The coding schedule should lay out the predicted completion dates for each module.

**10. Design the automated testing program into the application.** This provides a good source of information early in the development and testing phases.

**11. Design and start writing test code.**

**12. Perform regular code reviews.** Even a slight slip in schedule will cause developers to forget their responsibility toward the coding standard.

**13. Begin preparing for the beta program.** Design the beta application forms and choose a broad base of testers. Make sure proper communication channels are set up for defect reporting and product distribution.

**14. Complete the testing phase.**

**15. Establish a plan for having "Watson" meetings.**

**16. Release the application to beta.**

**17. Prepare the Golden Master Disks once all the defects are resolved.**

Since committing to a quality assurance program, MSR has dropped its defect rate considerably. Two projects can be cited as an example. Before QA measures were established, MSR's main product was approximately 60,000 lines of code. The defect count came to fifty-nine once defect tracking was started. The current product includes 75,003 lines of code and has a defect count of thirty-one.

With the addition of quality assurance procedures, MSR has seen an increase in profits and OEM contracts. The ability to deliver quality software has allowed the company to start extensive business relationships with two industry leading companies, one of which is a Fortune 100 company and the other a dominant peripheral supplier. By minimizing the effort of development in the software life cycle, MSR has begun to diversify its base of products. This is a crucial step for a company in its early years of growth.

**SOURCES**

(BEIZ90)
Beizer, Boris. 1990. Software testing techniques (2nd Edition). New York: Van Nostrand Reinhold.
(BROO75)
Brooks, Fredrick P. 1975. The mythical man-month. Reading, Massachusetts: Addison-Wesley Publishing Company.
(LEDG87)
Ledgard, Henry. 1987. Professional software, Software engineering concepts. Reading, Massachusetts: Addison-Wesley Publishing Company.
(MAGU93)
Maguire, Steve. 1993. Writing solid code. Redmond, WA: Microsoft Press.
(THIE92)
Thielen, David . 1992. No bugs!. New York: Addison-Wesley Publishing Company.
(SAND94)
Sanders, Joc and Eugene Curran. 1994. Software quality. New York: Addison-Wesley Publishing Company.

# A Software Quality Strategy for Demonstrating Early ROI

**Abstract**

Top management in many organizations won't make the investment or commitment to software quality without a cost benefit analysis or some demonstrated early results. To make matters worse, studies indicate that 80% of all quality efforts have no measurable results or improvement. One of the best ways to demonstrate early results is by piloting software inspections. Recent advances in software process improvement allow organizations to demonstrate early return on investment (ROI) within 6-12 months. If your organization is just starting out in quality, or you need to generate some ROI excitement in your current quality approach, this quality strategy may be just what you need.

**Biography of Timothy G. Olson**

Mr. Timothy G. Olson is a quality improvement consultant (both internal and external), a Juran Institute Associate, an Authorized SEI Lead Assessor, and a SEMATECH representative for software process improvement. While performing quality consulting, Mr. Olson has helped organizations measurably improve quality and save hundreds of thousands of dollars in costs of poor quality.

**Juran Institute Associate**

Based in Wilton, Connecticut, the Juran Institute is a worldwide leader in research, consulting, and training in the area of managing quality. Juran Institute Associates are experts in quality management and other disciplines who share their knowledge with Juran Institute, Inc. Mr. Olson works with the Juran Institute as a SEI Lead Assessor, as a CMM based software process improvement consultant, and as an instructor for software inspections training.

**CPI Software Quality**

Mr. Olson is an internal software quality consultant at Cardiac Pacemakers, Inc. (CPI) where he currently spends about 75% of his time. CPI makes quality pacemakers and defibrillators that depend upon life critical software. At CPI, Mr. Olson is responsible for software quality improvement efforts on pacemakers and defibrillators. Currently, Mr. Olson is piloting software inspections to improve software quality, and to demonstrate high returns on investment (ROI) within short periods of time.

**SEI Process Program**

Mr. Olson worked for the Software Engineering Institute for almost 7 years in the Software Process Program for directors such as Mr. Watts Humphrey and Dr. Bill Curtis. He was one of the co-developers of Software Process Assessments, and was co-author of the SEI report "Conducting SEI-Assisted Software Process Assessments" with Mr. Watts Humphrey. Mr. Olson also was a co-author of the SEI training course "Defining Software Processes"; a co-author of a new SEI handbook entitled "A Software Process Framework for the Capability Maturity Model", CMU/SEI-94-HB-1; and helped lead a SEI client from CMM Level 1 to Level 2.

**Author's Address**

Quality Improvement Consultants
Attn: Tim Olson
3082 Hamline Ave. N.
St. Paul, MN 55113
(612) 636-1570

# A Software Quality Strategy for Demonstrating Early ROI

13th Annual Pacific Northwest
Software Quality Conference (1995)

By Tim Olson, CPI
Quality Improvement Consultants (QIC)
Juran Institute Associate
Authorized SEI Lead Assessor





## Presentation Objectives

Describe motivation for quality strategies.

Describe successful software quality strategies.

Present a software quality strategy for demonstrating an early return on investment (ROI).

Describe how to choose the right strategy for your situation.

Answer any of your questions.

# Agenda

**The Quality Crisis**

**Summary of Managing for Quality**

**Software Quality Planning Strategies**

**Software Quality Improvement Strategies**

**Choosing the Right Strategy**

**Questions and Answers**



# The Quality Crisis

**The cost of poor quality:**

- **"In most companies the costs of poor quality run at 20 to 40 percent... In other words, about 20 to 40 percent of the companies' efforts are spent in redoing things that went wrong because of poor quality" (*Juran on Planning for Quality*, 1988, pg. 1)**

- **Phil Crosby's Quality Management Maturity Grid states that if an organization doesn't know it's cost of quality, it's probably at least 20%. (*Quality is Free*, 1979, pg. 38-39)**

# Quality Management Maturity Grid

| STAGE | SUMMARY | COQ |
|---|---|---|
| Certainty | "We know why we do not have problems with quality." | 2.5% |
| Wisdom | "Defect prevention is a routine part of our operation." | 8% |
| Enlightenment | "Thru management commitment and quality improvement we identify and resolve quality problems." | 12% |
| Awakening | "Is it absolutely necessary to always have quality problems?" | 18% |
| Uncertainty | "We don't know why we have problems with quality." | 20% |

• Adapted from "Quality is Free", Crosby, pps. 38-39





# Quality Lessons Learned

80% of all quality efforts have no measurable results.

Most failures in quality are due to a poor choice of strategy.

In order to choose a quality strategy wisely, managers need to know how to manage for quality.

# Agenda

The Quality Crisis

**Summary of Managing for Quality**

Software Quality Planning Strategies

Software Quality Improvement Strategies

Choosing the Right Strategy

Questions and Answers





# Analogy: Managing for Finance

| Managing for Finance | Managing for Quality |
|---|---|
| Financial Planning: Setting business goals; budgeting | Quality Planning: Setting quality goals; design in quality |
| Financial Control: Cost control; actual vs. planned | Quality Control: Planned vs. actual quality goals; taking action on difference |
| Financial Improvement: Cost reduction; mergers; acquisitions | Quality Improvement: Waste and rework reduction; eliminate & prevent defects |

- Adapted from "Juran on Leadership for Quality: An Executive Handbook", Juran, 1989.

# Juran's Definition of Quality

"Fitness for Use"

**Product Features that Meet Customer Needs**

- Provide customer satisfaction
- Create product salability
- Compete for market share
- Respond to customer needs
- Quality costs more

**Freedom from Deficiencies**

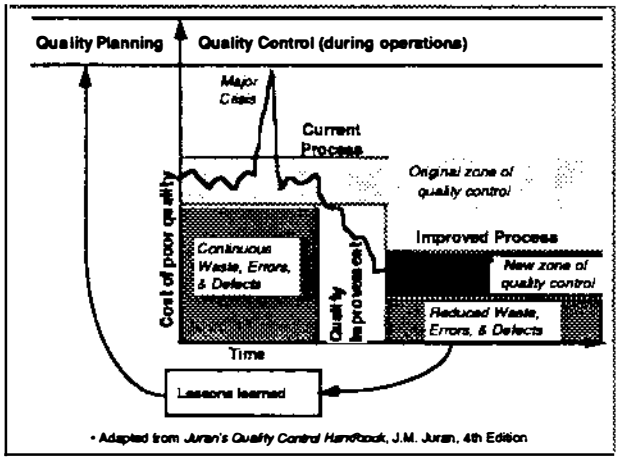
- Eliminate defects, errors, & waste
- Avoid product dissatisfaction
- Effect is on costs
- Quality costs less

• Adapted from "Facilitating Quality Improvement Teams", Juran Institute, Inc.



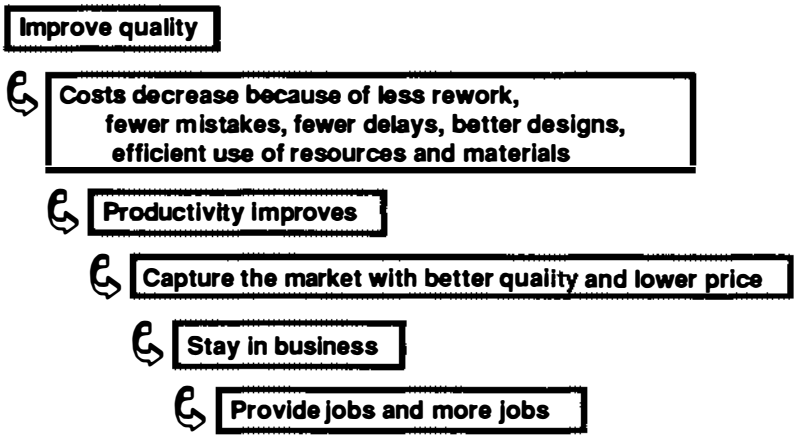


# The Juran Trilogy for Quality Management




• Adapted from *Juran's Quality Control Handbook*, J.M. Juran, 4th Edition

# The Quality Chain Reaction




• Adapted from W. Edwards Deming. *Out of the Crisis*. MIT, 1986, pg. 3





# Agenda

The Quality Crisis

Summary of Managing for Quality

Software Quality Planning Strategies

Software Quality Improvement Strategies

Choosing the Right Strategy

Questions and Answers

# Examples of Quality Planning

**Juran's Quality Planning Process**

**Quality Function Deployment (QFD)**

**SEI Software Process Improvement Model**
- **Strategy based on a maturity model**

**Software Project Management**
- **Use estimating techniques to improve cost performance index**





# Agenda

**The Quality Crisis**

**Summary of Managing for Quality**

**Software Quality Planning Strategies**

**Software Quality Improvement Strategies**

**Choosing the Right Strategy**

**Questions and Answers**

# Examples of Software Quality Improvement

Cost of Quality Estimates

Defect Prevention

Juran's Quality Improvement Process

Reviews and Walkthroughs

Software Inspections

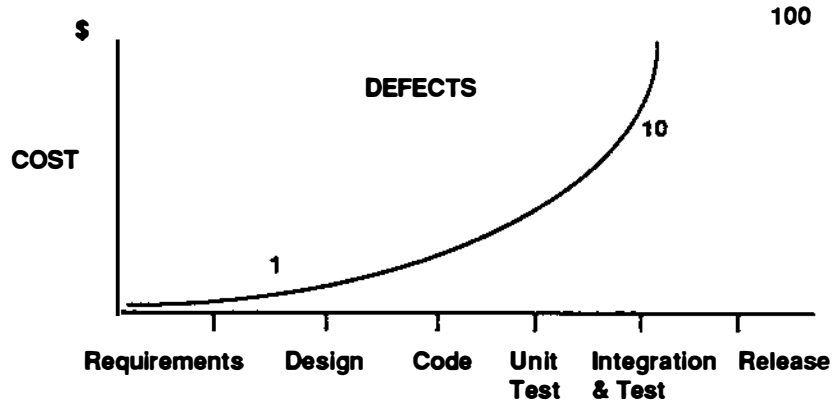
Software Testing





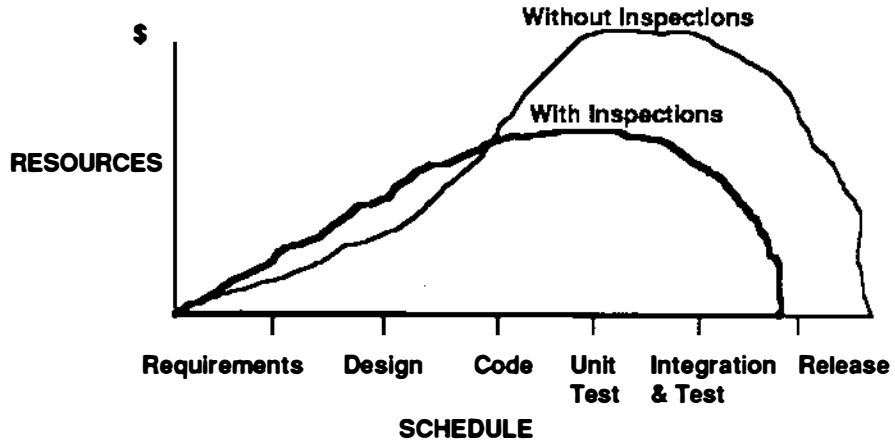
# Software Industry Standard Cost Ratio to Fix a Defect

Defects cost less to fix when detected earlier in the process




• Data from Gilb, T. and Graham, D. *Software Inspection.* Addison-Wesley, 1993.

# Inspections Shorten the Schedule




- Adapted from Fagan, M. "Advances in Software Inspections", IEEE Transactions on Software Engineering, July 1986





# Some Inspection Success Stories

**IBM reports [Fagan 86] [Ebenau 94]:**

- Inspections found 60-90% of all defects
- Productivity doubled

**NASA On-Board Flight Software [Billings 94]:**

- 85% of defects found before test phase
- Current defect rate .01 per KSLOC

**Jet Propulsion Laboratory (JPL) [Ebenau 94]:**

- Estimated $7.5 Million saved
- Corrective action reduced up to 90%
- Each inspection saves $25,000 average

**Hewlett Packard [Grady 94]:**

- Achieves and expects 10:1 ROI



- See references at the end of this presentation

# Key ROI Goal/Questions/Metrics

Goal: Measure ROI (both estimated and actual)

Key Questions:

1. How much does a defect cost in each phase of the software process (e.g., design vs. test vs. release)?

2. What is the defect removal rate of the verification processes for each phase (e.g., inspections, peer reviews, walk-throughs, testing)?

3. For a project's characteristics (KSLOC, staff, schedule, software process, etc.):
   - how many total defects (estimated and actual)?
   - how many total defects in each phase of the software process (estimated and actual)?



• See "A Software Quality Strategy for Demonstrating Early ROI", Olson, 1995



# Key ROI Metrics

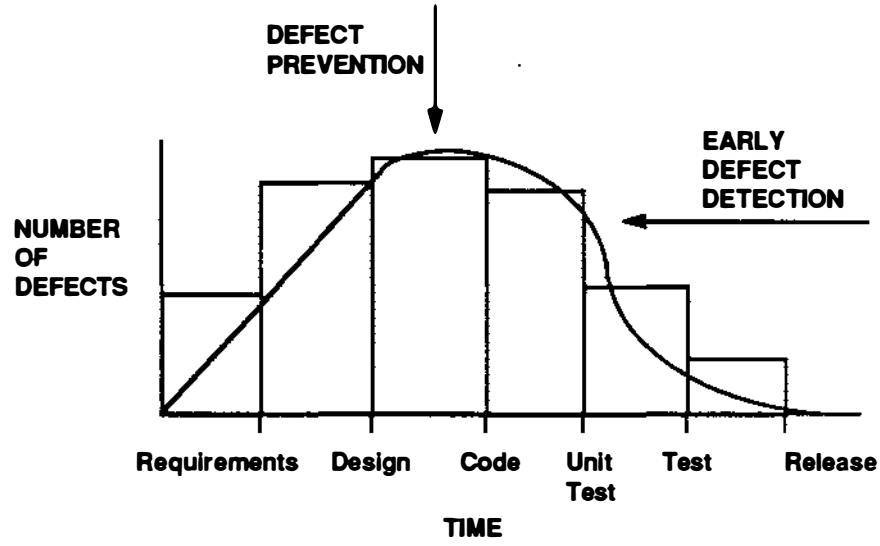
Key ROI metrics to compare inspection, peer review, walk-through, and test processes:

- Total percentage of project (effort or cost)
- Work product size by phase (total pages or KSLOC)
- Number of defects (total and by phase)
- Defect density (defects per page or KSLOC)
- Effort (person hours) per page or KSLOC
- Effort per defect (fully loaded process)
- Effort per defect (after defect is identified)
- Defect removal efficiency (by process and phase)
- ROI = Cost Reduction/Investment (factor in time)



• See "A Software Quality Strategy for Demonstrating Early ROI", Olson, 1995

# Key Quality Improvement Strategies






• See "A Software Quality Strategy for Demonstrating Early ROI", Olson, 1995



# Agenda

The Quality Crisis

Summary of Managing for Quality

Software Quality Planning Strategies

Software Quality Improvement Strategies

**Choosing the Right Strategy**

Questions and Answers

# Choosing the Right Strategy

| Strategies | Advantages | Disadvantages |
|---|---|---|
| Quality Planning | • Logically, the right 1st thing to do<br>• Most quality problems are planned that way<br>• Greater long term benefits | • Larger investment up front<br>• Measurable results take longer<br>• More difficult to sell to top management<br>• more difficult to implement successfully |
| Quality Improvement | • Early ROI<br>• Quality effort pays for itself early on<br>• Arouses greater enthusiasm<br>• Provides lessons learned to planning | • Systemic quality planning problems are not fixed<br>• Cheaper in the long run to redesign broken processes |



• See "A Software Quality Strategy for Demonstrating Early ROI", Olson, 1995



# Conclusion

Mature quality organizations use successful quality strategies.

Quality improvement strategies are a great way to start demonstrating early ROI (especially inspections).

In the long run, quality planning is best.

Effective prevention requires both quality planning and quality improvement.

# Agenda

The Quality Crisis

Summary of Managing for Quality

Software Quality Planning Strategies

Software Quality Improvement Strategies

Choosing the Right Strategy

Questions and Answers



# References


- [Barnard 94] Barnard, J. and Price, A. "Managing Code Inspection Information", IEEE Software, March 1994.
- [Billings 94] Billings, C. et al. "Journey to a Mature Software Process", IBM Systems Journal, vol. 33, no. 1, 1994.
- [Ebenau 94] Ebenau, B. and Strauss, S., *Software Inspection Process.* McGraw-Hill, 1994.
- [Fagan 86] Fagan, M. "Advances in Software Inspections", IEEE Transactions on Software Engineering, July 1986
- [Crosby 79] Crosby, P. B. *Quality is Free.* New York, NY: McGraw-Hill, 1979.
- [Deming 86] Deming, W. Edwards. *Out of the Crisis.* Cambridge, MA: MIT, Center for Advanced Engineering Study, 1986.
- [Dion 92] Dion, Raymond. "Process Improvement and the Corporate Balance Sheet." IEEE Software 10,4 (1992): 28-35.
- [Gilb 93] Gilb, T. and Graham, D. *Software Inspection.* Addison-Wesley, 1993.
- [Grady 94] Grady, R. and Van Slack, T. "Key Lessons in Achieving Widespread Inspection Use", IEEE Software, July 1994.
- [Juran 88a] Juran, J. M. *Juran on Planning for Quality.* N.Y., NY: Macmillan, 1988.
- [Juran 88b] Juran, Joseph. *Juran's Quality Control Handbook.* McGraw-Hill, 1988.
- [Juran 89] Juran, Joseph. *Juran on Leadership for Quality: An Executive Handbook.* New York, NY: Macmillan, 1989.
- [Olson 95] Olson, Timothy G. "A Software Quality Strategy for Demonstrating Early ROI", Society for Software Quality, May 1995.
- [Putnam 92] Putnam, L. and Myers, W. *Measures for Excellence.* Yourdon Press. 1992.

# SOFTWARE'S
# DIRTY DOZEN

This candid presentation looks at **the top twelve problems—and their solutions—that continue to plague software development projects**. Most projects today suffer from at least several of these problems, while some projects suffer from most of these problems. These problems focus on the areas of leadership, customer requirements, vendor relationships, development process, quality, project scheduling and tracking, and others. This presentation—based on the John Wiley & Sons' best seller, *Managing Software Development Projects: Formula for Success*—includes many real examples of projects and products (names withheld).

Vigorous competition and accelerating technological changes continue to demand shorter software development cycles. The challenge, however, is not only to develop products and solutions and distribute them to their markets faster, but also to produce products and solutions of increasingly higher quality at lower costs. It is the presenter/author's personal goal to "touch" as many people in the audience as possible in opening their eyes in not only understanding what the problems and their solutions are, but that *they can make a difference* in overcoming these problems in their organizations. Overcoming these major problems is critical to your company's—and therefore, *each project member's*—survival.

Neal Whitten is a popular speaker, trainer, consultant, and author in the areas of both project management and employee development. He has 25 years of front-line experience. While at IBM for 23 years, Whitten served in various management and leadership positions. He has managed the development of a variety of software products, including operating systems, business and telecommunications applications and special-purpose programs. He also managed an Assurance group providing independent assessments on numerous software projects. He is the author of John Wiley & Sons' best seller, *Managing Software Development Projects: Formula for Success*, copyright 1990; the Second Edition, copyright 1995, is now available. Whitten is also the author of *Becoming an Indispensable Employee in a Disposable World*, published by Pfeiffer & Company, copyright 1995. He is a member of the Project Management Institute and is a certified Project Management Professional. Whitten's services include that of project management trouble shooter, performing project reviews, and working with organizations in defining and streamlining software development processes. Whitten is President of The Neal Whitten Group.

Neal Whitten can be reached at The Neal Whitten Group, P.O. Box 858, Roswell, GA 30077-0858; 404-667-0881; fax 404-667-0588; Internet: nwhitten@ix.netcom.com.

# SOFTWARE'S
# DIRTY DOZEN
## OR

The Most Common
Problems That Plague
Software Development Projects

Neal Whitten



WHITTEN/SWDD001

## OPENING REMARKS

- About the Author
- "For Mature Audiences"
- Problems Are Widespread Across the Industry
- Solutions Are Offered
- Today's Project Practices

WHITTEN/SWDD001A

# Objectives

- To Change the Way You Think... In Recognizing and Doing Something About the Problems You Encounter in Your Job
- To Encourage You to "Make Things Happen"

WHITTEN/SWDD002

# Major Problem Areas

1. Leadership, Leadership, Leadership
2. Customer Requirements
3. Development Process
4. Quality
5. Product Ease of Use
6. Vendor Relationships

WHITTEN/SWDD003

## MAJOR PROBLEM AREAS *(cont.)*

7. Project Scheduling
8. Project Tracking
9. Managing Priorities
10. Employee Participation
11. Project Tools
12. Project Reviews

WHITTEN/SWDD004

## PROBLEM 1 : LEADERSHIP

- Unclear Definition
- Problem at All Skill Levels

- Little/No Formal Training
- Frequent Assignment Rotation

WHITTEN/SWDD005

## Attributes of the Successful Leader

- Create and Nurture a Vision
- Don't Fear Failure
- Expect and Accept Criticism
- Take Risks
- Empower Others... and Yourself

WHITTEN/SWDD006

## Attributes of the Successful Leader *(cont.)*

- Be Decisive
- Be Persistent
- Be Happy
- Be Serious About Humor
- Leave Your Ego Behind

WHITTEN/SWDD007

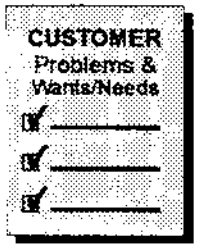
# Attributes of the Successful Leader *(cont.)*

- Challenge Conventional Thinking
- Meet Your Commitments
- Coach Your Team - Be a Role Model
- Maintain a Winning Attitude
- Believe in Yourself

WHITTEN/SWDD008

# Problem 2 : Customer Requirements

- Not Understood
- Not Fully Documented
- Agreement Not Obtained
- Not Verified
- Changes Not Sized




WHITTEN/SWDD009

# Problem 3 : Development Process

**Not Following an Acceptable, Defined Process**

- ☑ Built-in Checks & Balances
- ☑ Baselined Product Objectives and Specifications
- ☑ Easily Tailored
- ☑ Continuously Improved

WHITTEN/SWDD010

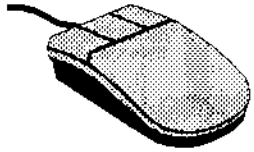
# Problem 4 : Quality

- Not Defined
- Not Measured
- Not Analyzed for Root Causes
- No Continuous Improvement Culture
- Targets Not Established


WHITTEN/SWDD011

# Problem 5: Product Ease of Use

- Treated as 2nd-Class Feature
- Product Objectives Missing:
  - Usability Functional Direction
  - User Tasks
  - Measurement Criteria

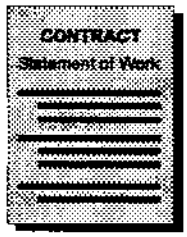


- Function Missing from Specifications
- Weak/No Usability Plan

WHITTEN/SWDD012

# Problem 6: Vendor Relationships

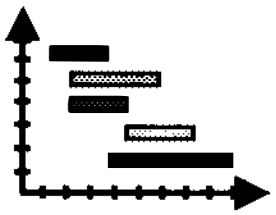
- Best Vendor Not Selected
- Incentives Weak or Missing
- Relationship Poorly Defined
- Commitments Weakly Tracked
- Untimely Resolution of Issues




WHITTEN/SWDD013

# PROBLEM 7 : PROJECT SCHEDULING

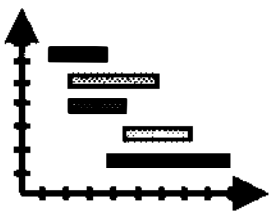
- Aggressive but Not Achievable
- In Place too Late
- Poor Planning of Document "Review Cycles"
- Poor Buffer Management
- Poor Planning of Overlapping Activities



WHITTEN/SWDD014

# PROBLEM 7 : PROJECT SCHEDULING *(cont.)*

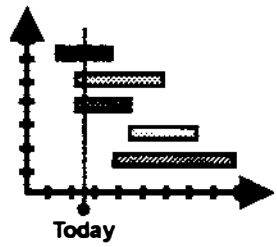
- Critical Paths Not Understood
- Top-Down Schedule Committed
- Commitments Weak or Missing
- No Second Opinion
- Change Too Often



WHITTEN/SWDD015

# PROBLEM 8 : PROJECT TRACKING

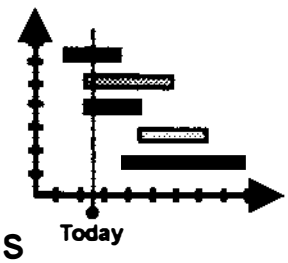
- Problems Discovered Too Late
- No Easy Method to Log Problems
- Infrequent/Irregular Tracking Meetings
- Inadequate Tracking Charts
- Inefficiently Run Tracking Meetings




WHITTEN/SWDD016

# PROBLEM 8 : PROJECT TRACKING *(cont.)*

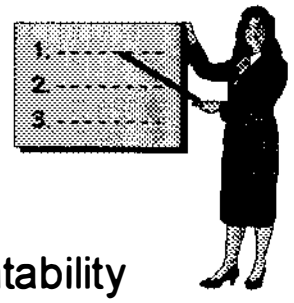
- Poor Dependency Management
- Late/No Distribution of Meeting Minutes
- Incomplete/No Recovery Plans
- Late/No Escalations to Resolve Issues
- Problems Not Tracked to Closure




WHITTEN/SWDD017

# PROBLEM 9 : MANAGING PRIORITIES

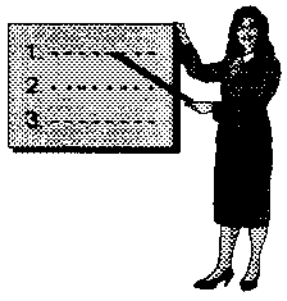
- Not Knowing What Are the Most Important Problems
- Not Working Most Important Problems First
- Not Assigning People Accountability for Resolving Problems




WHITTEN/SWDD018

# PROBLEM 9 : MANAGING PRIORITIES *(cont.)*

- Not Insisting on Trackable Plan to Resolve Problems
- Not Reviewing Top 3-5 Problems Daily
- Having Same Problems on "Hot List" Each Month




WHITTEN/SWDD019

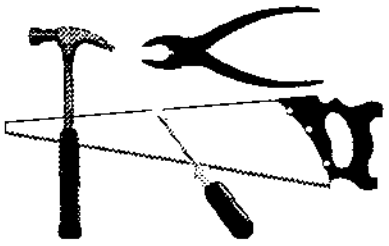
# Problem 10 : Employee Participation

- No Comprehensive Training of Development Process
- Not Driving Decision-Making Process Downward
- Not Encouraging Employees to Take Responsibility for Resolution of Project/Team Problems
- Weak/No Empowerment

WHITTEN/SWDD020

# Problem 11 : Tools

- Weak Investment in Acquiring/Developing State-of-the-Art Tools
- Not Standardizing on Proven Set
- No/Little Training for Project Members



WHITTEN/SWDD021

# Problem 12 : Project Reviews

- No Process/Quality Reviews Every 3-4 Months
- No Periodic Business Reviews
- No Post-Project Reviews

WHITTEN/SWDD022

# Think About It...

- No One Person or Group is Any More Important Than Another Person or Group within the Project
- Work as an Individual... and as a Team
- Make Manager a Hero
- Trust Your Instincts

WHITTEN/SWDD023

# THINK ABOUT IT... *(cont.)*

- Say No... Professionally
- You Are Responsible for Your Own Actions
- Your Reality Is What You Choose It to Be
- You Can Make the Difference in Ensuring Change

WHITTEN/SWOD024

# Using Fault-Injection to Assess Software Safety Standards

J. M. Voas, C. C. Michael
Reliable Software Technologies Corporation
Loudoun Tech Center
Suite 250, 21515 Ridgetop Circle
Sterling, Virginia 20166
phone 703.404.9293, 703.404.9294
fax 703.404.9295
{jmvoas, ccmich}@RSTcorp.com

K. W. Miller
Sangamon State University
Department of Computer Science
Springfield, IL 62794
phone 217.786.7327
miller@eagle.sangamon.edu

## Abstract

Fault-injection methods appear to have a useful ability to assess the safety of software as well as to demonstrate the effectiveness of software processes. This paper looks at how classes of anomalous behavior can be injected into executing software in order that the net impact of said injections can be observed. This observation allows for a prediction for whether the software satisfies different safety requirements.

## The Authors

Jeffrey Voas is the Principal Researcher at RST and heads the research initiatives of the company. In this capacity, Voas is currently the principal investigator on research initiatives for NASA, National Institute of Standards and Technology, National Science Foundation, and the US Air Force. Voas has coauthored a text entitled Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995). Dr. C. C. Michael holds a technical staff position at RST as a senior research scientist. As a graduate student, Michael designed and implemented the mutation analysis algorithms used in RST's commercial product, the *PiSCES Software Analysis Toolkit*$^{(TM)}$. He developed a simple and flexible interface language for specifying how code is mutated by *PiSCES*, as well as schema-based methods for runtime mutant evaluation. Dr. Michael also developed a significant portion of the source-code processing methods in the original release of *PiSCES*. Dr. Keith Miller is an associate professor of computer science at Sangamon State University. His research interests include software engineering, software reliability, and computer ethics. Miller received a PhD in computer science from the University of Iowa. Miller is a member of ACM and IEEE.

## Keywords

fault-injection, software standards, process models, software quality, simulated fault-injection, software safety assessment.

# 1 Introduction

Software standards should be closely related to software behavior and they should be quantifiable. These considerations are particularly important when standards are applied to critical systems. A quality-enhancement method must be shown to improve the dependability of a specific piece of software, not simply be touted as a generally "good thing to do."

If a measurement of software quality successfully predicts software behavior, that measurement can be also used to assess the effectiveness of a software process. When a process is successful, software developed using that process will be measured as having quantifiably better behavior. Behavior-based measurements act as a check on claims for process improvements.

Fault-injection methods are quantifiable and focus on studying software behavior. Fault-injection techniques can assess software and software quality standards in three ways:

1. help determine if a software standard results in higher quality software,
2. demonstrate that a particular piece of software satisfies a particular standard, and
3. help fulfill the process requirements of some standards.

In the remainder of this paper, we will discuss general fault-injection techniques and their application to software *safety-related* standards. Section 2 of this paper discusses fault-injection and introduces two specific fault-injection techniques: one designed to assess safety, and the other designed to assess testability. In Section 3, we discuss applying these two techniques to various requirements in three existing software standards:

1. NASA Interim Report NSS1740.13–process oriented software safety standard,
2. Underwriter's Laboratory Standard for Safety UL1998 entitled "Safety-Related Software" [13]–product oriented quality standard, and
3. FAA's DO-178B [2]–process oriented quality and safety standard.

# 2 Background

Engineering standards in software development are necessary on the most intuitive grounds: we must not use haphazard construction when building the most complex objects humans make. Good engineering requires quantification, and software quality is the most fundamental software characteristic. The *direct* measurement of adequate software quality (via testing) is impractical, and will remain so for the foreseeable future. This section presents an *indirect* measurement technique, *fault-injection*, that is practical for measuring certain attributes associated with software safety. When applied to collections of software produced with different development techniques, fault-injection may also help assess the effectiveness of software processes. This however requires large statistical samples that are independently taken, and hence needs years of evidence.

We propose to exploit "fault-injection" techniques to move beyond the practical limits of testing. Fault-injection techniques fall under the umbrella of processes that are called "fault-based." Fault-based techniques are designed to show either:

1. that a particular class of faults are not in a system, or
2. the impact some class of faults will have on a system if they exist in the software.

Fault-based processes can be harnessed to evaluate the quality of software engineering processes or the software itself.[1]

Fault-injection techniques have long been used on physical systems to test for *robustness*. For instance, an airplane manufacturer might freeze water on a wing in a wind tunnel to test the wing's lift under such conditions–here the manufacturer is not necessarily testing a problem with the wing, but rather how the wing will perform in undesirable circumstances. For software, we can "play" similar "what-if" games, except that we will additionally show that fault-injection methods can suggest at the quality of the development standards themselves.

The most widely used fault-based software methods deal with software testing. *Error-based testing* is a fault-based testing method that defines certain classes of errors and the subdomain of the input space which should reveal any error of that class if that error type exists in the program [11, 16]. Morell [9] proves properties about error-based strategies concerning certain errors that can and cannot be eliminated using error-based testing. Since error-based testing restricts the class of computable functions, it is limited as well. Morell's work does not inject classes of faults into a system.

*Fault-seeding* is an fault-based technique that estimates both the number of faults remaining as well as their type. Faults are seeded (injected) into a copy of the original software, and the "seeded" versions are run. Based upon the number of artificial faults discovered during testing, an estimate is made of the number of remaining actual ("*inherent*") faults [8]. A drawback is that if the seeded faults are not representative of the inherent faults, the estimate is invalid. *Stratified fault-seeding* [15] improves fault-seeding by showing that faults of a particular impact do not exist, instead of a syntactic class of faults. This is our first example of how fault-injection can assess the effectiveness of a testing process.

Mutation testing is a second example of a fault-based technique that can assess the effectiveness of a testing process. *Mutation testing* [12] is a particular error-based strategy that injects syntactic faults in code to evaluate candidates for test inputs. If a set of input points distinguishes the programs with the injected faults from the original code, then we assume that this set will find any actual faults in the original code.

To perform *true fault-injection*, the user must know that the modification to the code that they are making will cause failure, which requires some notion of an oracle or specification. It is possible that a modification will be a fix instead of a fault. In contrast, *simulated fault-injection*, makes changes to either the code or state of the executing program, without regard for what those changes actually represent. In fact, the changes might be fixes, although this is quite unlikely. The techniques we discuss here are simulated fault-injection, not true fault-injection.

In the next two sections, we look at two specific fault-injection techniques. The first is used to analyze software safety, and the second is used to analyze software testability. As already stated, we will emphasize the utility of the fault-based safety assessment technique.

## 2.1 Safety Analysis Based on Fault Injection

"Extended propagation analysis" (EPA) is the first fault-injection that we will consider; it is a simulated fault-injection technique that can assess both the safety of the product [5] and the quality of the "design-for-safety" process [6]. EPA collects *dynamic* information concerning which output variables are affected by a data state value that is "somehow" altered. EPA differs from conventional

[1]For process $A$ to stand as the judge for the value of process $B$ implies that process $A$ is of high enough quality to make such a determination; evaluating the quality of fault-based methods is out of the scope of this paper. Here, we assume that fault-based techniques are empirically capable of refereeing other processes.

fault-tree analysis because (1) a forward trace is made from an execution of the program, not from static analysis; and (2) EPA concentrates on data state propagation, not on software faults, when isolating regions of concern in the code. EPA isolates program regions that propagate certain types of data state errors allowing us to predict whether certain types of software failure will result.

EPA is related in purpose to *static* fault-tree analysis. In software safety, Leveson has written extensively on fault-tree analysis, and she has been involved in applying it to several different large programming projects [3]. Fault-tree analysis traces "backwards" in a cause/effect sequence from identified hazards we wish to avoid to conditions which would allow that hazard to occur. Fault-tree analysis is traditionally done in the early stages of design, but it can be done from code. In short, EPA is a natural extension of the basic hypothesis on which fault-tree analysis is based, however EPA is dynamic, code-based, and makes a forward trace.

### The Extended Propagation Analysis Theoretical Model

A *data state error* is an incorrect variable/value pairing in a data state where correctness is determined by an assertion between locations (statements).[2] We refer to a data state error as an *infection*, and use these two terms interchangeably. If a data state error exists, the data state and variable with the incorrect value at that point are termed *infected.* A data state may have more than one infected variable. *Propagation* of a data state error occurs when a data state error affects the output.

Let $S$ denote a specification, $P$ denote an implementation of $S$, $x$ denote a program input, $\Delta$ denote the set of all possible inputs to $P$, $Q$ denote the probability distribution of $\Delta$, $l$ denote a program location in $P$, and let $i$ denote a particular execution (or what we term an "iteration") of location $l$ caused by input $x$. And let $\mathcal{A}_{lPix}$ represent the data state produced after executing location $l$ on the $i^{th}$ execution from input $x$.

It is important to group data states into sets with similar properties. For instance, assume that location $l$ is executed $n_{xl}$ times by input $x$. Now consider all of the data states that are created by this input immediately before $l$ is executed or immediately after $l$ is executed. The following set allows us to do so:

$$\mathcal{A}_{lPx} = \{\mathcal{A}_{lPix} \mid 0 \leq i \leq n_{xl}\}$$

We further group these sets for all $x \in \Delta$:

$$\alpha_{lP\Delta} = \{\mathcal{A}_{lPx} \mid x \in \Delta\}$$

A *simulated infection* is a modified value that replaces the existing value of some variable in a data state. $\mathcal{A}_{lPix}$ denotes the data state created after the $i^{th}$ iteration of location $l$ on input $x$; $\breve{\mathcal{A}}_{lPix}$ denotes this same data state after a simulated infection is injected into $\mathcal{A}_{lPix}$. A simulated infection usually affects a single live variable.

It is important at this point to explain the relationship between simulated infections and the potentially disastrous states that a system can get into that can lead to a hazardous state. When a system gets into a "bad state" during execution, the next event that we would like to occur is recovery to an acceptable state. Otherwise we would like to demonstrate that the "bad state" has no damaging consequences. Simulated infections are the mechanisms that are employed in EPA to

[2] Admittedly, this is tenuous, since for any specification, there is an infinite number of correct programs that implement that specification, and therefore for each statement in a program version there must be an assertion if we are to expose data state errors, which as a practical matter will never happen. However, defining correctness at any granularity requires that we posit the possibility of such assertions.

allow observation of the impact of different classes of "bad states." Simulated infections mimic the effect of both programmer faults and hardware failures.

Simulated infections are created by perturbation functions. The process of injecting a simulated infection into an executing program is termed *perturbing.* A *perturbation function* is a mathematical function that takes in a data state as an incoming parameter, changes it according to certain parameters that are either input to the function or hard-wired, and produces as output a different data state. A data state that has had a value changed by a perturbation function is said to have been *perturbed.*

We consider that program $P$ has a fixed set of output variables: $\{v_1, v_2, v_3, ..., v_n\}$. An *affected variable* is an output variable whose value differs after a simulated infection is forced into the program, the program execution is resumed, and termination occurs.

EPA creates sets of affected variables that occur after some variable $a$ is perturbed on all iterations at location $l$ by the following algorithm. Since we are concerned about the fault-tolerance of a system regardless of whether the program experiences internal faults or corrupted incoming data, this algorithm is applied to every program variable.

**Algorithm:**

1. Set **k** to 0.
2. Set **variable_set** = $\emptyset$.
3. Increment **k**.
4. Randomly select an input $x$ according to $Q$, and if $P$ halts on $x$ in a fixed period of time, find the corresponding $\mathcal{A}_{lPx}$ in $\alpha_{lP\Delta}$. Set $\mathcal{Z}$ to $\mathcal{A}_{lP1x}$.
5. Alter the sampled value of variable $a$ found in $\mathcal{Z}$ creating $\check{\mathcal{Z}}$, and execute the succeeding code on both $\check{\mathcal{Z}}$ and $\mathcal{Z}$. If $l$ is executed more than once for $x$, i.e., $\mathcal{A}_{lP2x}$, ... $\mathcal{A}_{lPmx}$, alter $a$ in each $\mathcal{A}_{lPix}$, $2 \leq i \leq m$.
6. For each output variable that contains a different value (after comparing $P$'s output using $\check{\mathcal{Z}}$ to the output $P$ regularly produces), add it as a member to the set **variable_set**; place all undefined output variables into **variable_set** if a time limit for termination has been exceeded and the original program, $P$, terminated. This precaution is necessary because mutated variables can cause infinite loops.
7. Set $\Pi_{alPQk}$ = **variable_set**. ($\Pi_{alPQk}$ represents the set of output variables that have different values given execution of $P$ occurs with $\check{\mathcal{A}}_{lPx}$. $\Pi_{alPQk}$ is the empty set if no output variables are affected by the injection of $\check{\mathcal{A}}_{lPx}$.)
8. Repeat steps 2-7 $n$ times.

This algorithm produces the sets: $\Pi_{alPQ1}, \Pi_{alPQ2}, ..., \Pi_{alPQn}$. We then create a set of these sets:

$$\{\Pi_{alPQk} \mid \ 1 \leq k \leq n\}$$

This set represents all combinations of output variables that experienced different values when $a$ was perturbed at $l$. A similar algorithm is given in [7] that only adds to **variable_set** if the output

state caused by the corruption is a software hazard, and hence it is not necessary to execute a version of the program in which no data state corruption occurs.

It is important to recognize that there are two important considerations that are implicit in this algorithm:

1. the results are a result of randomly selecting inputs according to $D$, and
2. the results are a function of the perturbation function(s) that produces the altered value(s).

Therefore it is imperative that $D$ approximate the operational distribution as closely as possible, and the perturbation functions approximate the expected class of external anomalous events as closely as possible. The accuracy of these approximations directly affect the accuracy of the resulting predictions.

The time costs of performing an analysis such as this is directly related to $n$, the number of locations ($l$) where the perturbation functions are injected, the average number of fault classes applied at a single $l$, and speed at which the program runs for a single $x$. It is our experience that for critical programs that are typically less than 100 KSLOC and run quite quickly (milliseconds), the analysis can be completed in hours to days on an average quality workstation. Furthermore, the analysis is highly automatable, and parallelizable.

## 2.2 Using Extended Propagation Analysis To Detect "Dangerous" (Unsafe) Locations

We now take the information provided by EPA and produce the set of locations from which a particular type of software failure could result. Recall that the goal here is to identify where specific hazardous failures could originate. Note that if the analysis is applied in regions of *dead code*, the algorithm will never consider $l$ as having been reached, and hence the results will not indicate high fault-tolerance, but rather no results, suggesting dead code. The implementation that we use of this algorithm requires that $x$ reach $l$, and if this is never true, our tool produces a warning.

Let $\mathcal{V}_P$ represent a set of sets. Each member of $\mathcal{V}_P$ contains either a single output variable or a combination of output variables of program $P$. Each internal set represents one type of software failure of $P$. For instance, if $\mathcal{V}_P = \{\{v_1\}, \{v_2, v_3, v_4\}\}$, then we have identified 2 types of software failure: the first type occurs when the single output variable $v_1$ is incorrect, and the second type occurs when the output variables $v_2, v_3$, and $v_4$ are all incorrect. If we apply the EPA algorithm and

$$(\exists k \mid 1 \leq k \leq n)(v_1 \in \Pi_{alPQk})$$

we predict that if the value of variable $a$ at location $l$ is incorrect, output variable $v_1$ will be incorrect. Thus if location $l$ is incorrect and this "incorrectness" affects the value of $a$, the first failure type is predicted to occur. If

$$(\exists k \mid 1 \leq k \leq n)(v_2 \in \Pi_{alPQk}) \wedge (v_3 \in \Pi_{alPQk}) \wedge (v_4 \in \Pi_{alPQk})$$

we predict that the second type of failure will occur if location $l$ causes variable $a$ to be incorrect. Performing this analysis isolates locations that we predict can cause a class of software failure defined in $\mathcal{V}_P$. Dynamically, this reveals that there is a location $l$ that is not fault-tolerant for the classes of failure in $\mathcal{V}_P$.

This has a direct application to safety critical software. Let $\mathcal{C}_P$ represent a set of sets that is similar to $\mathcal{V}_P$ above. Each internal set in $\mathcal{C}_P$ contains either a single output variable or a combination of output variables of program $P$.[3] If either the single output variable or combination of output variables are ever corrupted, a hazardous event of the system that $P$ controls will result. When we apply the EPA and

$$(\exists k \mid 1 \leq k \leq n)[(\exists \gamma \mid \gamma \in \mathcal{C}_P) \ \ (\gamma \subseteq \Pi_{alPQk})] \tag{1}$$

we predict that if the value of variable $a$ at location $l$ is corrupted, critical software failure is possible. The $l$s, where Equation 1 is true, are code regions that warrant concern during the operational phase, since these regions have the potential to propagate unsafe data state errors into critical software failures. This is the set of locations that are candidates for additional fault-tolerant mechanisms.

Thus, EPA can be combined with the results of hazard analysis to demonstrate whether particular classes of *hazardous failure* are unlikely or impossible. In combination with hazard analysis, EPA is able to statistically predict that the software product is safe. But what if anything does this say about the design-for-safety processes that were employed? What it says is that for *this* particular product, the processes produced a high quality product. Note that this says nothing absolute about future products that result from an identical process. But if a sequence of products were built that were deemed safe and received the identical design-for-safety process at independent sites, then we would gain statistical confidence in the quality of the process.

### 2.3 Testability Analysis Based on Fault Injection

EPA, as described above, uses information about the propagation of injected data state corruptions to analyze software safety. The second fault-injection technique that we will consider is termed "Sensitivity Analysis [14]," which is similar to EPA, is useful for assessing the likelihood that the testing scheme can make errors observable (which is simply Voas's definition of testability [14]).

Sensitivity analysis is like EPA in that it measures propagation of perturbed data state infections; it is different from EPA in that it also measures the effect of syntactic injected faults (similar to mutation testing). In addition, sensitivity analysis includes execution statistics as part of its calculations. For the sake of brevity, we will not give a prolonged description of sensitivity analysis or its implementation; for further details, see [14].

Given the results of sensitivity analysis, you can either say that:

1. The product is or is not testable.
2. The testing process is or is not acceptably powerful at detecting errors [4].

Sensitivity analysis is used to assess how likely software is to reveal its faults during testing; in this regard, it is a quantifiable measure of software as a product. The same analysis can be used to assess the effectiveness of different testing strategies, giving information about a software development process.

## 3 Applying Fault-Injection to Three Software Standards

Three recently released software engineering standards were designed to improve the quality of safety-critical software systems. In this section we examine what role (if any) simulated fault-

[3] $\mathcal{C}_P$ is determined during hazard analysis in the requirements phase and is directly related to system safety requirements.

injection can play in assessing these standards. Two standards are process-oriented, and one is product-oriented.

## 3.1 NASA Interim Report NSS1740.13

The recent NASA Interim Report NSS1740.13 introduces a software safety standard that states that the Code Safety Analysis phase of development must:

> "Identify potentially unsafe states caused by input/output timing, multiple events, out-of-sequence events, failure of events, adverse environments, deadlocking, wrong events, inappropriate magnitude, improper polarity, and hardware failure sensitivities, etc."

EPA is capable of simulating instances of these potentially dangerous events (minus the "etc" which could include anything from what we can tell) via perturbation functions (Step 5 of algorithm), and then determining the impact (if any) on the output. It is up to the user to decide if such an impact is troublesome (from a *system* rather than software perspective) during hazard analysis. The empirical, dynamic nature of EPA makes this NASA software safety standard practical.

EPA can empirically demonstrate that the software product satisfies particular portions of the NASA standard. But what does this fault-injection scheme say about the effectiveness of the development processes employed? For a critical system, in which hazard analysis is almost certainly performed at some level of rigor, programming language fire-walls are used to protect critical sectors from non-critical sectors. EPA can empirically demonstrate whether the design-for-safety processes (such as static fault tree analysis) produced software adequately protected by these firewalls. That is, simulated fault-injection via EPA can test whether, *for this product*, the development process has produced a safe system. As EPA successfully validates a sequence of such products, it begins to archive evidence to validate the quality of the process that created them.

### 3.1.1 ADA and NASA NSS1740.13

Let's take several anomalies from the above excerpt and provide example perturbation functions needed for Step 5 of the EPA algorithm for a safety-critical system written in ADA83; these methods can be adapted for other languages. To use this algorithm and these perturbation functions to mitigate hazards, Step 6 of the algorithm must only add to **variable_set** if the output state created by $\check{\mathcal{Z}}$ is a hazardous output state. Here, it is unnecessary to execute $P$ with $\mathcal{Z}$.

1. **input/output timing errors** Timing errors can occur among tasks executing in parallel, or between a program and one or more external systems with which the program is supposed to be communicating in real time. We will discuss the first of these anomalies below under the heading of out-of-sequence events.

   Timing anomalies in real-time can result if an input or output event occurs later than expected, or if it occurs sooner than expected. The first case can be simulated by inserting a delay in front of the input or output event:

   ```
   Put(x);
   ```

   would be modified as follows:

   ```
   perturbed_delay(12);
   Put(x);
   ```

where `perturbed_delay` is a function that permits us to selectively delay, or fail to delay, at the point where it is inserted.[4] (The number 12 given as an argument to `perturbed_delay` is used to tell the function which code location it is being called from. In the example above, there are at least eleven other locations in the code where delays (or other perturbations) can be selectively inserted; the example shows the twelfth. The function `perturbed_delay` will only delay one location during any one execution of the code, so that we will know which location is responsible if a timing anomaly causes a fault.)

The actual code for perturb_delay is as follows:

```
procedure perturbed_delay(id: in integer) is

begin

     if id = perturb_target then
          delay(standard.duration(perturb_time));
     end if;
end;
```

Here perturb_target is a global value identifying the location to be delayed. This procedure is actually simplified somewhat; the actual implementation will be synchronized with a task that obtains `perturb_target` from a driver process that repeatedly executes the Ada program while perturbing different locations. Methods for determining how long delays should be are formalized in [1].

2. **inappropriate magnitude** The dynamic range-checking mechanisms of Ada imply that it may be appropriate to simulate these anomalies with two types of perturbations. The first type of perturbation will take the magnitude of the perturbed variable outside of its legal range, raising an exception. This will allow the safety of the exception-handling mechanism to be evaluated. The second will perturb the value of a variable with in it's legal range, but outside of the range that is appropriate for a given location.

   The first type of perturbation is appropriate when a variable is assigned a value. Thus, if the code contains the assignment

```
   a : small_integer;
   b : larger_integer;
         .
         .
 begin
         .
         .
   a := small_integer(b);
         .
         .
```

[4] To speed-up a process involves delaying other processes.

the modified code will be

```
a := small_integer(range_perturb(12, integer(b),
                          integer(larger_integer'first),
                          integer(larger_integer'last),
                          integer(larger_integer'first),
                          integer(larger_integer'last)+1));
```

('first and 'last are predefined attributes that give the smallest and largest legal values of a scalar type). The number 12 identifies the location in the code, as above. The code for range_perturb will be given below.

The second type of perturbation is appropriate when a variable is about to be used. Thus, if the types of a and b are as in the previous example, then

```
tilt_flaps(b);
```

would be modified as follows:

```
tilt_flaps(larger_integer(range_perturb(12, integer(b), smallest_appr,
                                  largest_appr,
                                  integer(larger_integer'first),
                                  integer(larger_integer'last))));
```

with the parameters smallest_appr and largest_appr supplying the smallest and largest appropriate magnitude for b in the context of this location.

The function range_perturb is declared as follows:

```
function range_perturb(location: in integer;
                       perturbed_value: integer;
                       minimum: integer;
                       maximum: integer;
                       min_pert:integer;
                       max_pert:integer)
         return integer is

        perturbation : integer;
        sign         : integer;

        pragma suppress(RANGE_CHECK);  --- make sure that any range-check
                                       --  are raised in the instrumented
                                       --  code, and not here.

begin
        if (location = perturb_target) then
                perturbation := random_integer(min_pert, max_pert);
                sign         := random_integer(0,1);
```

```
                if sign = 1 then
                        return maximum + perturbation;
                else
                        return minimum - perturbation;
                end if;
        end if;
end;
```

The values `min_pert` and `max_pert` are parameters giving the minimum and maximum deviation beyond the appropriate bounds that our perturbation will cause. Note that the function must be overloaded to allow the perturbation of floating point types. The function `random_integer`$(a, b)$ returns a random integer between $a$ and $b$, inclusive. (This integer will be drawn according to an equilikely distribution unless subsequent experience shows that other distributions are more likely to uncover unsafe conditions.)

3. **wrong events** The wrong events that are of interest, in the context of safety assessment, are those that are erroneously coded into the program. If a program's normal flow of execution during testing causes a wrong event to take place, the event's affect on safety can be assessed by observing the outcomes of the tests. Hence, if the affect of a wrong event on safety cannot be assessed through simple random testing, it follows that the program did not follow the flow of control that would have resulted in the wrong event.

   Therefore, to simulate wrong events, we perturb the program's flow of control. This is done by changing the conditions associated with conditional executions, loops. accept statements, and so on.

   The conditional

```
if <condition> then
      <sequence of statements>
end if;
```

   will be changed to (for example)

```
if boolean_control(12) xor ( <condition> ) then
         <sequence of statements>
end if;
```

   As above, the parameter 12 indicates that this is the twelfth candidate for perturbation, and is used to control which one of several possible perturbations is actually activated. The function `boolean_control` is simply:

```
function boolean_control(id: in integer) return boolean is

begin
        if id = perturb_target then
                return TRUE;
```

```
        end if;

        return FALSE;
end;
```

Since `boolean_control` and the original condition are exclusively or'd, the boolean condition in which it is embedded takes on the same value that it would take on if there were no code instrumentation if the function returns FALSE. If the `boolean_control` returns `TRUE`, the value of the boolean condition is the opposite of what it would otherwise have been. As with the other functions outlined in this document, `perturb_target` is a global variable, and the actual implementation of the function will contain code that ensures the value has been initialized before it is used.

Wrong events can also be include random perturbation of any variable and missing code faults. Perturbation functions to handle these classes of anomalies are provided in [14].

4. **hardware failure sensitivities** The simulation of hardware failure depends on the type of failure being simulated. It is common to ask what can happen if a program input, whose value is supplied by hardware, receives an incorrect value because of a hardware failure. Our approach for this case will be to perturb the value that has been input.

```
Get(x);
```

will be changed to

```
Get(x);
x := range_perturb(12, x, x, x, integer'first, integer'last);
```

where `range_perturb` is the same function that was used in the section on inappropriate magnitude. Called as above, `range_perturb` simply replaces `x` with a random integer within it's legal range.

5. **failure of events** The simulation of these events is similar to the simulation of wrong events. Instead of simply perturbing the flow of control, we modify it so that a particular condition cannot be true. The conditional

```
if <condition> then
        <sequence of statements>
end if;
```

will be changed to (for example)

```
if not boolean_control(12) and ( <condition> ) then
        <sequence of statements>
end if;
```

The function `boolean_control` is the same one that was described in the section discussing wrong events. When perturbation for location 12 is activated by making `boolean_control` return `TRUE` for that location, the condition in the `if` statement is always false.

6. **deadlocking** Deadlocking can be simulated by simply making a task unavailable for rendezvous. This can be done by inserting what amounts to an infinite delay:

```
loop
        delay 60.0;
end loop;
```

7. **improper polarity** The nature of improper polarity perturbations will vary depending on the definition of polarity for the data type being perturbed. A simple case is that in which a boolean value is perturbed. In that case

```
a : boolean;
b : boolean;

    .
    .
    .
begin
    .
    .
    .
a := b;

    .
    .
    .
end;
```

would (for example) be instrumented as

```
a : boolean;
b : boolean;

    .
    .
    .
begin
    .
    .
    .
a := boolean_control(12) xor b;

    .
    .
    .
end;
```

where `boolean_control` is the same function used for simulating wrong events. The perturbation simply inverts the value assigned to `a`.

8. **out-of-sequence events** These faults can come about as the result race conditions between tasks; such faults can be simulated by inserting code to delay the input/output activities of certain tasks with respect to those of other tasks to ensure that. For example, if the code

```
Put(x * y);
```

appears in the code, it can be perturbed as follows:

```
perturbed_delay(12);
Put(x * y);
```

where `perturbed_delay` is the function described above under the heading of I/O timing errors.

It is also possible for there to be an unexpected sequence of events within a single module. The simulation of this type of anomaly is identical to the simulation of wrong events, because the argument we have made for wrong events also holds for this type of out-of-order events.

The instrumentation for these eight classes of perturbation functions shows how fault-injection techniques could allow NASA to gain a perspective on the risks of their critical software with respect to their standard. By providing a capability that simulates instances of the classes of anomalies described in this standard, a plausible way of assessing compliance with the standard is provided.

In summary, EPA is capable of simulating anomalous events via its fault-injection mechanisms and determining the impact. It is up to the user to decide if such an impact is troublesome (from a *system* rather than software perspective), but given that EPA is totally automated in terms of the results bookkeeping and fault injection instrumentation, it is clear that this capability can be applied to NASA's newest software safety standard.

## 3.2 Underwriter's Laboratory UL1998

EPA can also demonstrate whether a system is resistant to several of the concerns voiced in Underwriter's Laboratory Standard for Safety UL1998 entitled "Safety-Related Software" [13]:

> "These requirements [UL1998] address risks that may occur as a result of faults caused by software errors, such as the following: .... b) Coding errors, including syntax, incorrect signs, endless loops, and the like; c) Timing errors that can cause program execution to occur prematurely or late; d) Induced errors caused by hardware failure; e) Latent errors that are not detectable until a given set of conditions occur;....."

Once again, this standard requires a forward trace from a problem occurring during execution. Both EPA and sensitivity analysis can dynamically and automatically demonstrate the risks (impacts) of instances of most (if not all) of these classes of problems in the same manner as described for NASA Interim Report NSS1740.13 by careful applications of perturbation functions (Step 5 of EPA algorithm).

### 3.3 RTCA DO-178B

RTCA DO-178B [2] is the set of requirements that must be in followed during development of commercial aircraft software. For brevity, we will look at one small part of this standard, its *unit testing* requirements. These requirements enforce particular classes of structural unit testing for different levels of software functional criticality (A-highest, B, C, D, E-lowest). An experiment is planned in 1995 to gain preliminary evidence as to how effective modified condition/decision coverage (MCDC) is at detecting condition faults. MCDC is the required unit testing coverage for the highest level of criticality, A. The experiment will compare MCDC to system-level testing, branch testing (unit), multiple condition coverage testing (unit), and condition/decision testing (unit), to judge its relative effectiveness at detecting errors with respect to its cost. If the study is unable to show that MCDC is substantially better, given that it cost substantially more, additional studies are anticipated. If enough evidence is collected that MCDC is not cost-effective, it is possible that this requirement of DO-178B could be relaxed. As before, the empirical nature of fault-injection focuses on behavior, not structure, and this allows fault-injection to be employed to compare fault detection effectiveness of different testing techniques.

## 4 Conclusions

This paper has briefly presented the underlying theory behind simulating faults in software. We have shown how simulated fault-injection helps assess standard compliance. We have argued for the application of fault-injection methods to three current safety-critical standards: [10], [13], and [2].

Fault-injection methods are capable of simulating large classes of anomalous software events, including programmer faults and corrupt external data. Interestingly enough, fault-injection methods are more precise when the class of anomaly being simulated is well-defined, e.g., in the situation where we know *a priori* how some external hardware device will fail if it fails. We are less likely to have this luxury when programmer faults are the anomalous event being simulated. Hence for standards where external anomalies are of serious concern, fault-injection techniques such as EPA have apparent application.

Simulated fault-injection performed in EPA and sensitivity analysis is empirical, not formal, and thus the results are not absolute guarantees of how the system will behave when deployed, because fault-injection methods make limiting assumption on the degree of anomalous circumstances that can occur at once. If greater numbers of difficulties were to be presented to a program simultaneously, then we suspect that there would be a greater likelihood of output corruption occurring. But if future anomalies are correctly represented by the perturbation functions, which are based on the standards, then the predictions are accurate. Both EPA and sensitivity analysis produce their predictions based on prior observations of software experiments. Although empirical, simulated fault-injection is not testing, and requires no oracle. This allows it to be fully automated, greatly enhancing its practical application to large software systems. To date, the largest systems that we have applied instrumentation to are on the order of 100 KSLOC.

In summary, fault-injection techniques measure software behavior using empirical techniques that deliver quantifiable results. These results measure software quality of products; repeated and independent use of these measurements may be useful in analyzing the effectiveness of the processes used to generate the software.

### Acknowledgement

The authors wish to thank the reviewer who provided many in depth insights and criticisms that have resulted in a much stronger paper. This research was partially funded by NASA Contract NAS1-20388.

## References

[1] Reliable Software Technologies Corporation. Quantifying Confidence in the Correctness of Parallel/Distributed Software. Technical report, Sterling, Virginia, USA, July 1993. Final Report for Contract NAS1-19896.

[2] Federal Aviation Authority. Software Considerations in Airborne Systems and Equipment Certification, 1994. Document No. RTCA/DO-178B, RTCA, Inc.

[3] N.G. Leveson and P.R. Harvey. Analyzing Software Safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[4] J. Voas and K. Miller. The Revealing Power of a Test Case. *J. of Software Testing, Verification, and Reliability*, 2(1):25–42, May 1992.

[5] J. Voas and K. Miller. Dynamic Testability Analysis for Assessing Fault Tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.

[6] J. Voas and K. Miller. An Automated Code-Based Fault-Tree Mitigation Technique. In *Proc. of 14th International Conf. on Computer Safety, Reliability, and Security*, Italy, October 1995.

[7] J. Voas and K. Miller. Examining Software Quality (Fault-tolerance) Using Unlikely Inputs: Turning the Test Distribution Up-side Down. In *Proc. of Eighth Annual Conference on Computer Assurance.*, National Institute of Standards and Technology, Gaithersburg, MD, June 1995.

[8] H. D. Mills. On the Statistical Validation of Computer Programs. *IBM Federal Systems Division, Report FSC-72-6015, Gaithersburg, MD.*

[9] Larry Joe Morell. A Theory of Error-based Testing. Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.

[10] Nasa. NASA Software Safety Standard. Office of Safety and Mission Assurance, June 1994. Interim Report 1740.13.

[11] E. J. Weyuker and T. J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, SE-6:236–246, May 1980.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[13] Underwriters Laboratory Inc. Safety Related Software, January 1994. Standard for Safety UL1998, First Edition.

[14] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.

[15] L. J. Morell and J. Voas. Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability. Technical Report WM-88-2, College of William and Mary in Virginia, Department of Computer Science, September 1988.

[16] S. J. Zeil. Testing for Perturbations of Program Statements. *IEEE Transactions on Software Engineering.* SE-9(3):335–346, May 1983.

# Predicting Fault-Prone Modules: A Case Study

Taghi M. Khoshgoftaar
Software Engineering Measurement Group
Dept. of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
Email: taghi@cse.fau.edu
Phone: (407)367-3994

## Abstract

Can software quality be predicted early in development? Early prediction of quality would enable developers to make improvements at a cost-effective point in the development life cycle. In this presentation we focus on faults as one aspect of software quality. In many situations, prediction of the number of faults is not really needed. Rather, identifying the most troublesome modules early may be sufficient. In such cases, classification models serve quite well.

We present a case study of a sample of modules representing about 1.3 million lines of code, taken from a much larger real-time telecommunications system. This study used discriminant analysis for classification of fault-prone modules, based on measurements of software design attributes and categorical variables indicating reuse history. We developed several models, demonstrating that certain features improve predictions.

We compared models based on lines of code only, on the best correlated metric only, and on multiple product metrics. We found the multivariate model made much better predictions.

Multivariate models can be misleading if the underlying metrics are highly correlated. Principal components analysis is a statistical technique for transforming data into uncorrelated variables. Our case study illustrates that principal components analysis can substantially improve the predictive quality of a software quality model.

The case study shows that reuse history can be used to improve quality models based only on design product metrics. We found that the model that included reuse data had substantially better predictive accuracy than the one that did not.

**Dr. Taghi M. Khoshgoftaar** is a member of the faculty of the Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Florida. An internationally known authority on software measurement, software reliability and quality engineering. He has also published more than 100 papers in these areas. He often is invited to serve on program committees, to referee papers, and to speak at symposia. For the years 1992 and 1994 he served as the Program Co-Chairman for the IEEE International Symposium on Software Reliability Engineering. He is Program Chairman of the 1995 Software Engineering Research Forum. He was a Guest Editor of the *IEEE Computer* – special issue on the *Metrics in Software*, September 1994. He is North American editor of the *Software Quality Journal*. Also, he is on the editorial board of the *Journal of Multimedia Tools and Applications*.

## Related Work


[1] T. M. Khoshgoftaar and E. B. Allen. Detection of fault-prone program modules in a very large telecommunications system. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995. IEEE Computer Society. Forthcoming.

[2] T. M. Khoshgoftaar and E. B. Allen. Multivariate assessment of complex software systems: A comparative study. In *Proceedings of the First International Conference on Engineering of Complex Computer Systems*, Fort Lauderdale, FL, Nov. 1995. IEEE Computer Society. Forthcoming.

[3] T. M. Khoshgoftaar, E. B. Allen, and A. de Gramont. Identifying change-prone telecommunications software modules during testing and maintenance. In *Proceedings of the Annual Oregon Workshop on Software Metrics*, Silver Falls, OR, June 1995. Oregon Center for Advanced Technology Education, Portland State University.

[4] T. M. Khoshgoftaar, B. B. Bhattacharyya, and G. D. Richardson. Predicting software errors during development using nonlinear regression models: A comparative study. *IEEE Transactions on Reliability*, 41(3):390–395, Sept. 1992.

[5] T. M. Khoshgoftaar and D. L. Lanning. Are the principal components of software complexity data stable across software products? In *Proceedings of the International Symposium on Software Metrics*, pages 61–72, London, UK, Oct. 1994. IEEE Computer Society.

[6] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya. A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. *IEEE Journal on Selected Areas in Communications*, 12(2):279–291, Feb. 1994.

[7] T. M. Khoshgoftaar, J. C. Munson, B. B. Bhattacharya, and G. D. Richardson. Predictive modeling techniques of software quality from software measures. *IEEE Transactions on Software Engineering*. 18(11):979–987, Nov. 1992.

[8] T. M. Khoshgoftaar and P. Oman. Software metrics: Charting the course. *Computer*, 27(9):13–15, Sept. 1994. Guest editors of special issue on software metrics.

[9] T. M. Khoshgoftaar and R. M. Szabo. Improving code churn predictions during the system test and maintenance phases. In H. A. Müller and M. Georges, editors, *Proceedings of the International Conference on Software Maintenance*, pages 58–67, Victoria, BC Canada, Sept. 1994. IEEE Computer Society.

[10] T. M. Khoshgoftaar and R. M. Szabo. Investigating ARIMA models of software system quality. *Software Quality Journal*, 4(1):33–48, Mar. 1995.

[11] T. M. Khoshgoftaar, R. M. Szabo, and P. J. Guasti. Exploring the behavior of neural network software quality models. *Software Engineering Journal*, 10(3):89–96, May 1995.

[12] T. M. Khoshgoftaar, R. M. Szabo, and J. M. Voas. Detecting program modules with low testability. In *Proceedings of the International Conference on Software Maintenance*, Nice, France, Oct. 1995. IEEE Computer Society. Forthcoming.

[13] D. L. Lanning and T. M. Khoshgoftaar. Modeling the relationship between source code complexity and maintenance difficulty. *Computer*, 27(9):35–40, Sept. 1994.

[14] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

[15] R. M. Szabo and T. M. Khoshgoftaar. An assessment of software quality in a C++ environment. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995. IEEE Computer Society. Forthcoming.

## Overview

"Why software metrics?"
"How will it help me?"
Myths
Validation Principles
Case Study
From Lab to Practice
Beyond Detection of Fault-Prone Modules

## "Why software metrics?" Common Wisdom

Software Product Metrics
⇓
Amount of Information
⇓
Ease of Understanding
⇓
Reliability and Quality

## "Why software metrics?" Defining "Fault-Prone"

**Fault-Prone** is a quality factor.

An **incident** occurs during operation.
**A failure** is an incident due to a fault.
**A fault** is a defect in an executable product.
**A defect** is a result of an error.
An **error** is made by a person.
**Fault-Prone** is the class of modules
that exceeds a given threshold
for the number of faults discovered
during a period of time.

## "Why software metrics?" Using a Quality Model

Software Measurements
⇓
Quality Model
⇓
Predicted Quality Factor
⇓
Management Action

## "Why software metrics?" Example Results

Which modules are fault-prone?

Model made prediction for each module.

| Obs | Actual | Prediction | Pr(not f-p) | Pr(f-p) |
|---|---|---|---|---|
| 1 | not f-p | not f-p | 0.96 | 0.04 |
| 2 | not f-p | not f-p | 1.00 | 0.00 |
| 3 | not f-p | f-p | 0.00 | 1.00 |
| 4 | not f-p | not f-p | 0.71 | 0.29 |
| ... | | | | |
| 15 | not f-p | f-p | 0.18 | 0.82 |
| ... | | | | |
| 640 | f-p | not f-p | 0.99 | 0.01 |
| 641 | f-p | f-p | 0.02 | 0.98 |
| 642 | f-p | f-p | 0.00 | 1.00 |
| 643 | f-p | not f-p | 1.00 | 0.00 |
| ... | | | | |
| 660 | f-p | f-p | 0.00 | 1.00 |

## "Why software metrics?" Types of Misclassifications

| | | Predicted Class: not f-p | Predicted Class: f-p |
|---|---|---|---|
| Actual Class | not f-p | Ok | Type I Error |
| | f-p | Type II Error | Ok |

## "Why software metrics?" Example Results

Case Study Misclassification

| Actual | Error Rate | Error Count |
|---|---|---|
| Not fault-prone | 23.8% | (138/580) |
| Fault-prone | 13.8% | (11/ 80) |
| Overall | 22.6% | (149/660) |

Mistaking a **not fault-prone** module
as **fault-prone** wastes effort
looking for problems that are not there.
Missing a **fault-prone module**
risks problems after deployment.

"How will it help me?"
## Classification may be enough

Regression models predict a quantity.
Classification models predict class membership.
Management action may be class-oriented.

"How will it help me?"
## Supervisors

Work assignments
— Match skills to module complexity
Discover faults before system testing
by targeting fault-prone modules.
— More intense reviews
— More participants in reviews
— Extra reviews

"How will it help me?"
## Managers

More effective testing
by planning for fault-prone modules.
— Extra tests
— More thorough set of test cases
— Schedule allows for diagnosis
Avoid crisis during maintenance
by more realistic planning.
— Schedule/staff allows for problems expected

## Myth:

"LOC is enough."
Many software metrics are correlated to LOC.
LOC is easy to collect.

Myth:
## "LOC is enough."

If product has uniform characteristics
then LOC may be enough.
Many systems are not uniform.

Case study LOC misclassification

| Actual | Error Rate | Error Count |
|---|---|---|
| Not fault-prone | 22.6% | (131/580) |
| Fault-prone | 56.3% | (45/ 80) |
| Overall | 26.6% | (176/660) |

Error rate for fault-prone modules was
**much higher** than multiple metrics model.

Additional metrics often add accuracy.

## Myth:

"Just break up fault-prone modules."
Small is better.

Myth:
## "Just break up fault-prone modules."

Arbitrary break up will just shift complexity
from measured to unmeasured.
e.g. Internal vs. Interface complexity

## Myth:

"Just fire whoever makes
fault-prone modules."
We don't want programmers on our team,
who make poor quality software.

Myth:
## "Just fire whoever makes fault-prone modules."

Best programmers are often assigned
to modules with most difficult requirements.
Some complexity is justified by function.

Fault-prone is not necessarily
poor quality.

## Validation Principles

"Valid metric must have a useful
statistical relationship to a Quality Factor."
— Schneidewind

Validation case study must be realistic
— No toy problems
— No student programmers
— Large enough for team of programmers
— Real world application

Predictive validation case study
— Simulates use of quality model.
— Correlation is not enough
— Fit model with data from one system
— Predict with data from another system

## Case Study
## System Description

| | |
|---|---|
| Application | Telecommunications |
| Language | Pascal-like |
| Lines of Code | 1.3 million |
| Executable Statements | 1.0 million |
| CFG Edges | 364 thousand |
| Source Files | 25 thousand |
| Functional Modules | Sample of 2 thousand |

Faults were discovered from Coding through Operations.
56% of modules had no faults due to reuse.

## Case Study
## Metrics

Quality Factor: Faults
High Level Design Metrics:
- Modules Used
- Total Direct Calls
- Unique Direct Calls

Detailed Design Metrics:
- Non-loop Conditional Arcs
- Loops
- Nesting Level Total
- Span of Conditional Arcs
- Vertices+Arcs within Loop Spans
- McCabe cyclomatic complexity, $V(G)$

## Case Study
## Reuse Covariates

Delta variables were differences between current and prior version measurements
New modules had no prior version.

$$ISNEW = \begin{cases} 1 & \text{All delta variables are missing} \\ 0 & \text{Otherwise} \end{cases}$$

Reused modules had no changed measurements.

$$ISCHG = \begin{cases} 0 & \text{All delta variables are zero} \\ 1 & \text{Otherwise} \end{cases}$$

$ISNEW$ and $ISCHG$ were additional variables in model.

## Case Study
## Method

Given data
1. Standardize to mean=0 and variance=1.
2. Principal components analysis.
3. Split data into *Fit* and *Test* data sets.
4. Select significant variables of *Fit*.
5. Estimate model parameters based on *Fit*.
6. Evaluate quality of fit.
7. Predict Fault-Prone modules in *Test*.
8. Compare predictions to actual.

Model ready to apply to a current similar project.

## Single vs. Multivariate Models
## $LOC$ Lines of Code

If product has uniform characteristics
then LOC may be enough.
Many systems are not uniform.

Pr(Fault-Prone) = $f(LOC)$

| | | Predicted Class: not f-p | Predicted Class: f-p |
|---|---|---|---|
| Actual Class | not f-p | 77.4% | **22.6%** |
| | f-p | **56.3%** | 43.7% |

Overall misclassification: 26.6%

## Single vs Multivariate Models
## Modules Used

Modules Used had the highest correlation to Faults.

Pr(Fault-Prone) = $f(ModsUsed)$

| | | Predicted Class not f-p | Predicted Class f-p |
|---|---|---|---|
| Actual Class | not f-p | 98.6% | **1.4%** |
| | f-p | **80.0%** | 20.0% |

Overall misclassification: 10.9%

## Single vs Multivariate Models
## Design Metrics

Input: 9 raw design metrics

Pr(Fault-Prone) = $f(MU, SpCond, Nest)$

| | | Predicted Class not f-p | Predicted Class f-p |
|---|---|---|---|
| Actual Class | not f-p | 62.8% | **37.2%** |
| | f-p | **22.5%** | 77.5% |

Overall misclassification: 35.5%

## Single vs Multivariate Models

Let "Design Metrics" mean 9 CG and CFG metrics.
Type II error rates for single metrics were **much higher** than multiple metrics models.

Additional metrics often add accuracy.

## Principal Components Analysis
## Domain Pattern

| Metric | Domain 1 | Domain 2 | Domain 3 |
|---|---|---|---|
| SpLoop | **0.901** | 0.359 | 0.137 |
| Loops | **0.880** | 0.370 | 0.134 |
| SpCond | **0.719** | 0.545 | 0.316 |
| Nest | **0.683** | 0.593 | 0.334 |
| Tot Calls | 0.359 | **0.864** | 0.216 |
| Uniq Calls | 0.426 | **0.830** | 0.245 |
| VG | 0.597 | **0.724** | 0.309 |
| If Then | 0.599 | **0.681** | 0.357 |
| Mods Used | 0.177 | 0.265 | **0.939** |
| Eigenvalues | 3.63 | 3.41 | 1.46 |
| % Variance | 40.3% | 37.9% | 16.2% |
| Cumulative | 40.3% | 78.1% | 94.4% |

## Principal Components Analysis
## Design Components

Input: 9 CG and CFG metrics
PCA retained $D_1, \ldots, D_3$

Pr(Fault-Prone) = $f(D_1, D_2, D_3)$

| | | Predicted Class not f-p | Predicted Class f-p |
|---|---|---|---|
| Actual Class | not f-p | 67.6% | **32.4%** |
| | f-p | **21.25%** | 78.75% |

Overall misclassification: 31.1%

## Principal Components Analysis

Multicollinearity **may** cause unstable parameters.
Stable parameters yield more robust model.
PCA stopping rule reduces dimensionality by ignoring noise in data.

PCA avoids problems due to multicollinearity.

## Reuse Covariates
## Design Metrics

Input: 9 raw design metrics with covariates
Only a few metrics were significant at 5%.

Pr(Fault-Prone) =
$f(MU, SpCond, Nest, ISNEW, ISCHG)$

| | | Predicted Class not f-p | f-p |
|---|---|---|---|
| Actual Class | not f-p | 66.2% | **33.8%** |
| | f-p | **16.25%** | 83.75% |

Overall misclassification: 31.7%

## Reuse Covariates

Reuse information improves
accuracy of quality models.

## Early Prediction
## Design Components

Input: 9 CG and CFG metrics
PCA retained $D_1, \ldots, D_3$

Pr(Fault-Prone) = $f(D_1, D_2, D_3, ISNEW, ISCHG)$

| | | Predicted Class not f-p | f-p |
|---|---|---|---|
| Actual Class | not f-p | 76.2% | **23.8%** |
| | f-p | **13.75%** | 86.25% |

Overall misclassification: 22.6%

## Early Prediction

Design metrics model gives
similar results as product metrics model,
but earlier.

## Summary of Models

Error Rates

| Model | not f-p | f-p |
|---|---|---|
| LOC | 22.6% | 56.3% |
| Mods Used | 1.4% | 80.0% |
| Raw | 37.2% | 22.5% |
| PCA | 32.4% | 21.3% |
| Raw,Reuse | 33.8% | 16.3% |
| PCA,Reuse | 23.8% | 13.8% |

## From Lab to Practice
## Technology Transfer

Data analysis
Proof of concept
— Metrics
— Models
— Methods
Prototype tools
Production tools

## From Lab to Practice
## Tools

Problem resolution system
— Incident → Failure → Fault
— Which module?
Configuration management system
— Source code library
— Versions
— Changes
Metric analyzer
Model building tools
— Statistics package
— Neural Net package
— Other AI packages
Model implementation tools

## Beyond Detection of Fault-Prone Modules
## Industry-Academia Partnership

Industry
— Practical perspective and priorities
— Full scale process and products
— Application of new technologies
Academia
— Research projects
— Quick response consulting
— Publications

# Metrics Measuring Control Flow Complexity in Concurrent Programs

Michael E. Mathews and Shengru Tu

Department of Computer Science
University of New Orleans
New Orleans, Louisiana 70148, USA
(504)286-7108, -7228 (fax)
Email: mmathe@nomvn.lsumc.edu, shengru@cs.uno.edu

## ABSTRACT

Software has rapidly evolved in the direction of parallel and distributed systems. However, software metrics have lagged behind especially in the area of concurrent software systems. In this paper, we have proposed a number of control flow metrics such as the task communication count (TC), the task conditional communication count (TCC), the forward reachable count (FC), the backward reachable count (BC), and forward chain length (FCL). These metrics can be applied in the design, implementation and maintenance stages of software development. Typically, metrics TC and TCC measure or predict the sheer workload in the implementation of different tasks, metrics FC and BC indicate the importance of program components, and FCL predicts the difficulty in tracing the execution of programs.

Theoretical evaluations of the proposed metrics have been conducted. The positive results of the theoretical evaluation have led us to a number of hypotheses of using our metrics. We believe that the proposed metrics can be used to measure or predict the effort for design, coding, testing and debugging with respect to synchronization structures between tasks. Experiments conducted with a number of well-understood programs of various sizes and types have illustrated phenomena consistent with our hypotheses. However, the testing and refinement of these hypotheses will be a long term work.

**keywords**: software complexity metrics, control flow complexity, concurrent programming, Ada tasking.

**Michael E. Mathews** received the MS degree in computer science from the University of New Orleans in 1994. Currently, he is a programmer analyst at LSU Medical Center, New Orleans, LA. His interests include distributed systems, software evaluation, and database systems.

**Shengru Tu** received the PhD dregree in electrical engineering and computer science from the University of Illinois at Chicago in 1991. He has been an assistant professor in the department of computer science at the University of New Orleans since 1991. Dr. Tu has published many research papers on analysis of concurrent programs. Currently, he is also director of a Louisiana State funded project, "computer based instruction to promote literacy". His research interests include analysis of concurrent programs, verification of distributed software, Petri net theory and applications, software metrics, and applications of multimedia technology.

# Metrics Measuring Control Flow Complexity in Concurrent Programs

Michael E. Mathews and Shengru Tu
Department of Computer Science, University of New Orleans

## 1. Introduction

Software complexity metrics can be defined as the measurement of the degree of difficulty in the stages of the software life cycle. These metrics can be used to support management in development of complex software by predicting critical information about reliability and maintainability of software systems, providing continuous feedback for ongoing designs, and pinpointing areas of potential instability for testing [McCabe94]. In fact, complexity measurement is one of the measures of software quality; given two functionally equivalent programs, we prefer to use the simpler one.

Software has rapidly evolved in the direction of distributed systems. Unfortunately, software metrics have lagged behind, especially in the area of concurrent software systems. Early metrics dealt explicitly with measuring the complexity of sequential programs [Cote88]. Software metrics for sequential systems are being used without special consideration of concurrency in systems. As a result, the reliability of these models may decrease to a point where they are no longer useful. Shatz has addressed this problem by proposing a Petri net graph model be used to determine concurrent software complexity [Shatz88]. The work in [Damerla92] focuses on the complexity due to nondeterminism in a concurrent system. While this is one property of complexity interest, we believe there are many more.

In this paper, we derive a number of additional metrics that measure control flow complexity of concurrent programs. Control flow metrics, also known as logic structure metrics, deal with the execution flow of a program. Concurrency in software systems adds a new dimension of complexity to software. As a result, complexity metrics should incorporate measurements of the various properties of a distributed system, including nondeterminism, communication, and synchronization. Ada tasking has been chosen as a vehicle to carry out concepts of concurrent programming, although the proposed metrics are conceptually independent of underlying languages.

## 2. Development of Basic Metrics and Derived Metrics

### 2.1 Basic Metrics

The main area of concern for concurrency complexity metrics are the communication points, namely entry call and accept statements in programs. We define the *task communication count*, TC($i$), of task $i$, as the number of communication points in task $i$. The use of the TC metric is based on the belief that the number of communication points contributes to complexity due to concurrency. We define the *program communication count*, PC, as the sum of all TC(i) in a program. This metric indicates the sheer number of communication points without considering any interaction between them.

Next, we consider the involvement of local control flows in communication schemes. We define the *task conditional communication count* for task $i$, TCC($i$), as the number of (separate) conditional branches within task $i$ that contain at least one communication point. If a branch of a conditional construct contains an accept or entry call statement, then that branch is counted in TCC. If a communication point appears in a nested conditional construct, then the number of levels of nesting is counted in TCC. In the case of conditional communication constructs, such as Ada's selective wait and conditional entry call constructs, every instance of a communication statement within such a construct is counted in TCC (e.g., every accept statement within a select construct is counted in TCC). Thus, the

TCC metric takes into account the local control flow of a task that tends to complicate communication. For a complete program, we define the *program conditional communication count*, PCC, as the sum of all TCC(i) in the program. This metric reflects the extent to which local control flows affect communication.

As a simple example of the TC, PC, TCC, and PCC metrics, we apply them to a program shown in Example 2.1. Because we measure control flow complexity due to task communication, the considered structures are skeletons containing control flow constructs that are involved in task communication. At this time, we view the complexity due to communication for this design as simply the number of communication points and the control flow structure within the tasks (i.e., the presence of conditional branches and loops). These two properties are measured by the TC and TCC metrics for individual tasks, and by the PC and PCC metrics for programs. The values of TC and TCC for each task is listed in Table 2.1.

**Example 2.1 A concurrent program design with three tasks**

```
task body T1 is
begin
    accept E1 do
        accept E2;
    end E1;
    accept E3;
end T1;
```

```
task body T2 is
begin
    accept E1;
    T1.E1;
    select
        accept E2 do
            T1.E3;
        end E2;
    or
        accept E3;
    end select;
end T2;
```

```
task body T3 is
begin
    T2.E1;
    T1.E2;
    if CONDITION then
        T2.E2;
    else
        T1.E3;
        T2.E3;
    end if;
end T3;
```

**Table 2.1**

| tasks | T1 | T2 | T3 |
|---|---|---|---|
| TC | 3 | 5 | 5 |
| TCC | 0 | 2 | 2 |

The intuition behind the TCC and PCC metrics is that the interruption of a sequence of one or more communication statements by a conditional branch containing one or more communication statements increases the complexity of a task in terms of overall program design (task structure), coding, testing, and debugging. During coding, programmers must consider how the placement of communication statements affect data flow within a task or program. Programmers who are given one or more tasks to code within a large system containing numerous interacting tasks must see that their tasks properly handle the passing of the correct data. For testing and debugging, the control flow of tasks and programs play an important role as each conditional path within a task or program must be checked. Communication points within conditional paths increase the effort needed to understand a task or program due to the addition of synchronization and possible transfer of data.

In the guidelines given by the Software Productivity Consortium [SPC89] to reduce conceptual communication complexity, they suggest that the following be minimized: (1) the number of accept and select statements per task, and (2) the number of accept statements per entry. The comparison of the following segments of Ada code illustrates their points. Our task-related metrics reflect the reduction of complexity by using the former code segment instead of the later code segment.

| Good | Bad |
|---|---|
| accept A;<br>if MODE_1 then<br>-- do one thing<br>else -- MODE_2<br>-- do something else<br>end if; | if MODE_1 then<br>accept A;<br>-- do one thing<br>else -- MODE_2<br>accept A;<br>-- do something else<br>end if; |

Next, we present more complexity measurements that take into account interactions between communication points and tasks.

## 2.2 Derived Metrics

Before deriving additional metrics, we first consider how a task affects the control flow of another task through communication. In Ada, two tasks can communicate with each other by rendezvous between an entry call statement and an accept statement. A task that executes an entry call statement actively affects the control flow of a called task, because when a task reaches to an accept statement, it cannot progress until another task makes a call to this waiting entry.

A task may affect the control flow of another task indirectly through a third task. For example, a task, $T_1$, has an entry call to an entry, E, in task $T_2$ and the control flow of $T_2$ leads from the statement "accept E" to a statement in $T_2$ that calls an entry in $T_3$. In this case, the control flow of task $T_1$ indirectly affects the control flow of task $T_3$. In Example 2.2, task A directly affects task B and indirectly affects task C. Note that although task C directly affects task D, task A does not indirectly affect task D, because there is no possible control flow from "accept C1" to "D.D1" in task C.

**Example 2.2 Tasks communicating directly and indirectly.**

```
task body A is
begin
   B.B1;
   B.B2;
end A;
```

```
task body B is
begin
   loop
      select
         accept B1;
         C.C1;
      or
         accept B2;
      end select;
   end loop
end B;
```

```
task body C is
begin
   D.D1;
   accept C1;
end C;
```

```
task body D is
begin
   accept D1;
end D;
```

Considering the above observation, we derive metrics indicating the amount of influence that a task imposes on other tasks. For an entry call statement *e*, in task *i*, we define the *forward reachable communication point count*, FC(*i, e*), as the number of *remote* communication statements that can be reached in the program's control flow. For an accept statement *a*, in task *i*, we define the *backward reachable communication point count*, BC(*i, a*), as the number of *remote* communication statements that can reach *a* in the program's control flow.

To formally define FCs and BCs, we use a model called the *entry call diagram*. An entry call diagram is a directed graph. Let G=(V, A) be such a graph, in which V is the vertex set and A is the arc set. Every vertex corresponds to a communication statement or a control flow statement such as *if* and

*loop* statements. Those vertices that correspond to entry call statements or accept statements are communication points. The local control flow of each task forms a control flow diagram. The arcs corresponding to local control flows are *local arcs* (*l-arcs*). From every entry-call vertex to the called accept-vertex, there is an arc. We call this kind of arc a *calling arc* (*c-arc*). An entry call diagram is the composition of a number of local control flow diagrams connected by c-arcs. An entry call diagram of the program in Figure 2.2 is shown below, where c-arcs are solid and l-arcs are dashed.

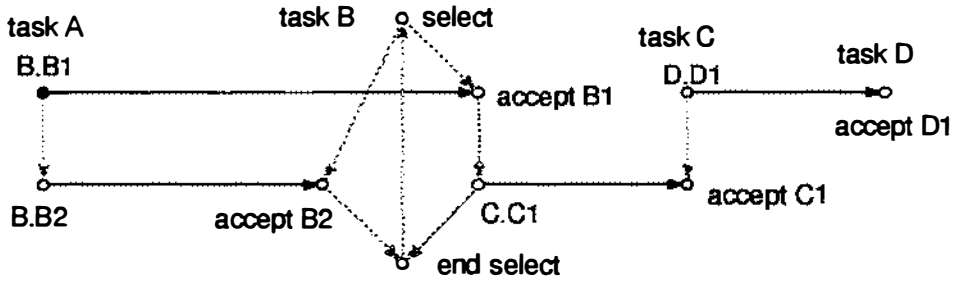



**Figure 2.3 An entry call diagram of the program in Example 2.2**

Using the entry call diagram, for an entry call statement $e$, in task $i$, the vertex subset consisting of all the reachable remote communication points from $e$ can be computed in $O(n)$ time, where $n$ is the number vertices in the diagram. This vertex subset is denoted by $FS(i, e)$. That is,

$$FS(i, e) = \{v \mid e \rightarrow v, \text{ v is a remote communication point}\}$$

where "$e \rightarrow$ v" means that there is a *simple* directed path (without duplicated vertices) from $e$ to v and this path does not contain any l-arc of task $i$ itself. Our first derived metric $FC(i,e)$ is defined based on $FS(i, e)$.

$$FC(i, e) = | FS(i, e) |$$

where $|\{...\}|$ denotes the cardinality of a set. Similarly, for an accept statement $a$, in task $i$, $BS(i,a)$ denotes the vertex subset consisting of all the remote communication points that can reach $a$, that is,

$$BS(i, a) = \{v \mid a \leftarrow v, \text{ v is a communication point}\}$$

where "$a \leftarrow$v" means that there is a directed path from v to $a$ and this path does not contain any l-arc of task $i$ itself. Based on $BS(i, a)$, we define another derived metric, $BC(i,a)$.

$$BC(i, a) = | BS(i, a) |$$

$FC(i, e)$ indicates the number of communication statements that $e$ can actively affect in a direct or indirect way. $BC(i, a)$ indicates the number of communication statements that can affect $a$ in a direct or indirect way. In practice, measurements FCs and BCs can be used in unit- or module-testing. We believe that more effort in program inspection should be put on those entry call statements with high FC values. On the other hand, more tests should be carried out focusing on those accept statements with high BC values, especially when testing resources (such as time) are limited.

However, FCs and BCs of statements are too fine to be used for design and system-level testing. Therefore, we define the FC and BC metrics of a task as the following.

$$FC(i) = | \cup_{e \in \text{task } i \text{ and } e \text{ is an entry call stmt}} FS(i, e) |$$

and

$$BC(i) = | \cup_{a \in \text{task } i \text{ and } a \text{ is an accept stmt}} BS(i, a) |.$$

The FC and BC of a task indicate the sheer amount of involvement of this task with other tasks. Table 2.1 lists the values of FC and BC for each task of the program in Example 2.2

**Table 2.1 The measurements of Example 2.2**

| tasks | FC | BC |
|---|---|---|
| A | 4 | 0 |
| B | 1 | 2 |
| C | 1 | 3 |
| D | 0 | 1 |

In addition to measuring completed programs, our metrics can also measure tasking skeletons representing large concurrent systems produced at early stages of design, as suggested by Byrne [Byrne91] and Gomaa [Gomaa94]. For management purpose such as dispatching manpower in design, testing or debugging, normalized FCs and BCs can be used to indicate the *relative importance* of each task in a software system. We define

$$FC\%(i) = FC(i) / n_c \qquad \text{and} \qquad BC\%(i) = BC(i) / n_c$$

where $n_c$ is the total number of communication points (entry call and accept statements altogether) in the system. By this normalization, FC% and BC% will be in the range of 0 and 1. FC% and BC% can also indicate the *role* of a task.

If a software system follows a commonly used design pattern such as the client-server model, better and easier understanding of this program can be expected. In a pure client-server system, server tasks provide entries (accept statements) only and have no entry call statements, and client tasks do not have any entries (accept statements). Thus, the FC of every server task is zero and the BC of every client task is zero. A high value (close to 1.0) of FC%($i$) indicates that task $i$ is an "aggressive client" in the sense that it directly or indirectly calls for service from a large part of the system. A high value of BC% implies that this task is essential to the entire system in the sense that a large part of the system calls for service from this task directly or indirectly. When a software system consists of tasks with either FC% close to 0 or BC% close to 0 (according to a tasking skeleton produced by a high level design), then a programming team having experience with general client-server systems can develop this system without difficulty. But when many tasks have FC%s and BC%s close to 0.5, programmers are not likely able to take advantage of their experience in client-server models. When some change is made locally in a task, FCs and BCs of other tasks will not change, but FC% and BC% will change. This is reasonable since these two metrics are used to indicate the relative roles of tasks in a system.

Although a high FC% or BC% value indicates the *importance* of a task, it may not necessarily pinpoint complex communication structures. For example, a simple server task may be called by every other task, which results in a high value of BC (close to 1.0). As another example, a task may call many simple servers sequentially, which will have a high value of FC. However, neither case is difficult to handle. We believe that long chains of causally related synchronization activities are the structures that complicate concurrent programs. Consequently, we define another metric, the *forward chain length* (FCL) for an entry call statement as follows:

$$FCL(i, e) = \max_{v\_is\_a\_commu.\_point} \{ \| e \rightarrow v \| \}$$

where "$e \rightarrow v$" again denotes a path from $e$ to $v$ without l-arcs in task $i$ and "$\| e \rightarrow v \|$" means the "*length*" of the path $e \rightarrow v$, that is, the number of remote communication points in the path. Note that vertices corresponding to local control structures are not counted. The FCL of a task is the highest FCL value of all the entry call statements in the task, i.e.,

$$FCL(i) = \max_{e \in task\_i} \{ FCL(i, e) \}$$

The FCL metric can be used to pinpoint potentially troublesome structures. Let us consider a simple example of a resource sharing system, in which users A and B *alternately* use the resource, and user A

should use it *first*. An inexperienced programmer may make a design in a very "natural" way shown as below.

```
task body User_A is
begin
loop
    Resource.E;
    User_B.Go;
end loop;
end User_A;
```

```
task body Resource is
begin
loop
    accept E do ... end E;
end loop;
end Resource
```

```
task body User_B is
begin
loop
    accept Go;
    Resource.E;
end loop;
end User_B;
```

This program marginally works. A potential problem is that user A may continuously grab the resource twice, although it will not be able to skip over user B twice. When we measure FCL, we find that FCL(User_A) = FCL(User_A, "User_B.Go") = 4. When the ratio FCL/$n_c$ is as large as 4/5=0.8, we have reason to reconsider the design. It turns out that the following design is simpler and enforces the desired ordering strictly, in which entries E1 and E2 in task Resource provide the same resource. The maximal FCL is now reduced to 2.

```
task body User_A is
begin
loop
    Resource.E1;
end loop;
end User_A;
```

```
task body Resource is
begin
loop
    accept E1 do ... end E1;
    accept E2 do ... end E2;
end loop;
end Resource
```

```
task body User_B is
begin
loop
    Resource.E2;
end loop;
end User_B;
```

## 3. Theoretical Evaluation of the Metrics

Evaluation of a software metric involves verifying that the metric measures what it is supposed to measure. In this section, we perform a theoretical evaluation of our metrics by showing that they satisfy most of the properties proposed by Weyuker [Weyuker88], which have successfully been used in theoretical evaluation of many other metrics such as McCabe's cyclomatic complexity measure [McCabe76], Halstead's composite measure [Halstead77], and Oviedo's data flow metric [Oviedo80]. Weyuker views a program as an object made up of *program bodies*. A recursive definition of a program body is given in [Weyuker88]. However, intertask synchronization was not considered in Weyuker's definition. We have evaluated our task-related metrics by augmenting Weyuker's original definition of a program body with tasking and communication statements.

### 3.1 Program Bodies

Since our metrics are defined to measure the complexity of synchronization/communication structures, we shall ignore purely local operations such as assignment statements. Borrowing the scheme given in [Weyuker88], we define a program body as follows.

**Definition -- Program Bodies**

(1) a task-interaction statement (an entry call or an accept statement) is a program body;

(2) a statement of the following form is a program body

**if** PRED **then** P **else** Q **end;**

where PRED is a predicate, and P and Q are program bodies;

(3) a statement of the following form is a program body

**if** PRED **then** P **end;**

where PRED is a predicate and P is a program body;

(4) a statement of the following form is a program body

**while** PRED **loop** P **end loop;**

where PRED is a predicate and P is a program body;

(5) one program body following another forms a program body

P; Q

(6) a selective wait statement or a conditional select statement is a program body.

In the above definition, cases (2) through (5) are identical to Weyuker's definition. In (1), we have replaced "assignment statement" in [Weyuker88] with "task-interaction statement". Case (6) is added to cover Ada's tasking features. This definition has augmented Weyuker's original definition of a program body with syntactically correct tasking and communication statements. Given an Ada tasking program, a synchronization skeleton satisfying the above definition can be extracted by filtering out local operations not involving any task interaction such as assignment statements.

### 3.2 Weyuker's Properties

In this section, we apply Weyuker's properties to the evaluation of our metrics. Although we have changed the definition of program bodies, we do not alter the intuition behind the original properties. For each property we assign a simple catch-phrase (taken from [Levine93]). Following each property, we evaluate our metrics. In the following, ||P|| denotes the complexity of a program body P as given by the metric under evaluation. Similarly, ||Q|| denotes the complexity of Q.

**Property 1**: *metric inequality*. By this property, a complexity measure which rates all programs as equally complex is not really a measure. That is, $(\exists P)(\exists Q)\ (\|P\| \neq \|Q\|)$.

Clearly, all of our metrics satisfy this property.

**Property 2**: *sufficiently fine*. This property reflects the intuition that a measure is too coarse if it divides all programs into just a few complexity classes, as Weyuker writes:

Let *c* be a nonnegative number. Then there are only finitely many programs of complexity *c*.

We find that our metrics satisfy this property.

**Property 3**: *sufficiently coarse*. Similar to the intuition of Property 2, a measure that assigns a unique complexity to every program is too fine, as Weyuker states:

There are two distinct programs P and Q such that $\|P\| = \|Q\|$.

It is easy to see that our metrics satisfy this property.

**Property 4**: *implementation dependency*. The intuition behind this property is that even though two programs calculate the same function, it is the details of the implementation that determine a program's complexity. An implementation dependent metric reflects how a computation is performed rather than what computation is being performed. That is, $(\exists P)(\exists Q)(P \equiv Q \ \&\ \|P\| \neq \|Q\|)$.

All of our metrics satisfy this property.

**Property 5**: *monotonicity*. The intuition behind this property is that for any measurement of syntactic program complexity, the components of a program are no more complex than the program itself. Weyuker states this as: $(\forall P)(\forall Q)(\|P\| \leq \|P;\ Q\|$ and $\|Q\| \leq \|P;Q\|)$, where P and Q are components, and P;Q denotes the concatenation of Q to the end of P.

Again, all of our metrics satisfy this property.

**Property 6**: *context sensitivity*. This property deals with Weyuker's definition of program bodies and the fact that concatenating them may not affect complexity in a uniform manner. Weyuker writes this as:

6a: $(\exists P)(\exists Q)(\exists R)(\|P\| = \|Q\| \ \& \ \|P; R\| \neq \|Q; R\|)$,

6b: $(\exists P)(\exists Q)(\exists R)(\|P\| = \|Q\| \ \& \ \|R; P\| \neq \|R; Q\|)$.

The TC and PC metrics fail to satisfy this property because they do not regard the context of communication statements in the complexity measurement. Based on the definitions of the remaining metrics, and considering a program or task to be composed of program bodies, this property holds for them.

**Property 7**: *statement order responsitivity*. This property asserts that program complexity should be responsive to the statement order and hence the potential interaction among statements. Weyuker states this as:

> There are program bodies P and Q such that Q is formed by permuting the order of the statements of P, and $\|P\| \neq \|Q\|$.

The TC and PC metrics fail to satisfy this property because they do not regard the position of communication statements in the complexity measurement. For the remaining metrics, their definitions naturally cause them to satisfy this property.

**Property 8**: *renaming insensitivity*. This property considers the relationship between the renaming of identifiers and the complexity of a program. Weyuker states this as:

> If P is a renaming of Q, then $\|P\| = \|Q\|$.

All of our metrics satisfy this property since they are dependent on structure only and do not respond to changes in identifier names.

**Property 9**: *component interaction sensitivity*. Weyuker asserts that, in some cases, the complexity of a program formed by concatenating two program bodies is greater than the sum of their individual complexities. This property is given by: $(\exists P)(\exists Q)(\|P\| + \|Q\| < \|P; Q\|)$.

The TC, TCC, PC and PCC metrics fail to satisfy this property because they do not regard the interaction of communication statements in the complexity measurement. FC, BC and FCL satisfy this property for program bodies communicating between different tasks. If we consider a program composed of tasks P and Q having one or more joining c-arcs, then the metric values of the isolated tasks will always be 0. Allowing the tasks to interact will produce a nonzero value.

The results of our evaluation are summarized in Table 3.1.

**Table 3.1 Theoretical evaluation summary**

| Property | PC | PCC | TC | TCC | FC | BC | FCL |
|---|---|---|---|---|---|---|---|
| 1) metric inequality | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 2) sufficiently fine | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 3) sufficiently coarse | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 4) implementation dependency | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 5) monotonicy | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 6) context sensitivity | No | Yes | No | Yes | Yes | Yes | Yes |
| 7) statement order responsitivity | No | Yes | No | Yes | Yes | Yes | Yes |
| 8) renaming insensitivity | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 9) component interaction sensitivity | No | No | No | No | Yes | Yes | Yes |

## 4. Hypotheses and Experimental Measurements

### 4.1 Hypotheses

The results of the theoretical evaluation have led us to a number of hypotheses of using our metrics as diagnostic indicators of efforts related to intertask synchronization in software development. It is commonly accepted that development of a concurrent program needs more effort than that for developing an equivalent sequential program. We call this part of additional effort the *concurrency related effort* that can be measured in person-hours. The metrics proposed in this paper are supposed to measure the concurrency related effort of developing concurrent programs. Specifically, first, we believe that TC and PC as well as TCC and PCC indicate the concurrency related effort in design and coding, since TC and PC measure the size of the synchronization structure and TCC identify complexity due to the involvement of communication in local control flows of each task.

**Hypothesis 1 -- Efforts for Design and Coding**

The concurrency related effort of developing programs is proportional to TC and PC and increases at quadratic or higher order when TCC and PCC increase.

Among the above four metrics, we surmise TCC and PCC reflect the effort in understanding control flows of concurrent programs. The larger the TCC and PCC values, the greater chance of misunderstanding which leads to faults. Furthermore, we believe that FLC is a sharper measure of the complexity of the synchronization structure in the sense that a small increment of a task's FLC value will cause a large increment of difficulty in verifying and debugging this task.

**Hypothesis 2 -- Efforts for Comprehension, Testing and Debugging**

The concurrency related effort for understanding, testing and debugging of a concurrent program increases at a quadratic or higher order when the FCL of this program increases.

When testing a program with complicated synchronization structures, how can we focus our attention to the portion of the program that will most likely catch potential errors? Branch coverage is a strong criterion for testing [DeMillo87]. It seems that tests on the task and the statements with high value of FCs may lead to a larger coverage of the program, even though FCs are derived from entry call diagrams which ignore data flow. We suspect that testing a program body having a high FC value will detect concurrency related errors with more likelihood than a test on a program body with a lower FC value in concurrent programs.

**Hypothesis 3 -- Coverage of Control Flows**

In tests on a task, the FC of this task is proportional to the average of actual coverage over the control flows of the entire program system.

Finally, we postulate that a high BC value of a task results in a greater chance of this task being affected by an error in other tasks. After a change of the synchronization structures, we believe that the tasks with high values of BC should be tested with priority.

**Hypothesis 4 -- Probability of Being Affected**

The probability that a task is affected by a change in the synchronization structure is proportional to the task's BC value.

To test the above hypotheses, extensive experiments should be carried out as part of software metric development process. We now perform measurements on five well-understood programs of various sizes and types. The primary approach is to compare the values of the metrics to the judgment obtained by reviewing programs that have been well understood.

### 4.2 Readers and Writers Program

In the readers and writers program, *r* reader tasks and *w* writer tasks access a shared buffer concurrently. When a writer is using the buffer, no other task can use the buffer, although multiple readers are allowed to access the buffer when no writer is using the buffer. This program has been proposed as a standard concurrent program to illustrate new metrics [Perlis81]. We employ a simplified version that avoids a steady stream of writers from blocking readers [Gehani91]. The program is shown in the appendix.

The measurements for the program are given in Table 4.1. The fact that FC(R_W)=0 and BC(READER_1) = BC(READER_2) = 0 indicates that this is a typical client-server system. Since task R_W enforces mutual exclusion, it is inherently more complex than the reader and writer tasks; hence the higher TC and TCC values of the R_W task predict more efforts in implementation as compared to the reader and writer tasks. R_W has a BC% close to 1.0 indicating that the management task R_W is very important. But in terms of control flow, it does not affect other tasks. For such a pure client-server system, its concurrent complexity is not high. Reflecting this, the FCLs of READER and WRITER tasks are small and do not change when *r* and *w* increase.

**Table 4.1 Measurements of the Readers-Writers Program**

| Task | TC | TCC | FC | BC | FCL |
|---|---|---|---|---|---|
| R_W | 5 | 5 | 0 | 2r+2w | 0 |
| READER_1 | 2 | 0 | 5 | 0 | 3 |
| WRITER_1 | 2 | 0 | 5 | 0 | 3 |

If we eliminate the for-loop in task R_W, all the nonzero FCs, BCs and FCLs will decrement by 1 only. Retrospectively, the use of this for-loop along with the accept statements in its iteration body is a good strategy in the sense that it successfully solves the problem and does not make the program much more complicated.

### 4.3 Gas Station Program

The client-server structure is not penicillin for every system. The gas station program simulates a system which is more complex than pure client-server systems [Helmbold85]. In a gas station, *n* customers (tasks) wish to buy gas at a self-service gas station. The station has some gas pumps (tasks) and an operator (task) that controls the pumps. To buy gas, each customer must prepay the operator before pumping the gas. When a pump is available, the operator activates the pump so that the customer can pump gas up to the amount prepaid. After pumping the gas, the customer gets change from the operator for the amount that was prepaid but not pumped. Next, the operator deactivates the pump and waits for the next customer.

We use a version of this program having one pump and two customers as shown in the appendix. None of the tasks is purely a client or a server. Correspondingly, the FC% and BC% of each task is between 0.6 and 0.84. The measurements of TC and TCC in Table 4.2 indicate that task Operator is the most complicated and involved one. The two customer tasks are simple but involved in the sense that they have a large influence on other tasks. Thus more effort in debugging may be required for the customer task. The equally high FC, BC and FCL values of the four tasks show that interaction between tasks is complicated.

Table 4.2 Measurements from the Gas Station program

| Task | TC | TCC | FC | BC | FCL |
|---|---|---|---|---|---|
| Operator | 10 | 8 | 15 | 15 | 6 |
| Pump | 7 | 6 | 18 | 18 | 9 |
| Customer_1 | 4 | 0 | 21 | 21 | 8 |
| Customer_2 | 4 | 0 | 21 | 21 | 8 |

### 4.4 Larger Example Programs

We have also applied our metrics to two larger programs, a merge-sort program with 4 tasks and a game simulation program with 15 tasks for a border defense system which have approximately 2,000 lines and 11,000 lines of Ada code respectively. Neither of them follows a pure client-server model. The measurements of our metrics have made it possible for us to divide the tasks into four groups: simple, simple but involved, complex but not involved, complex and involved. This classification is consistent to that made by a code review. We have made the tasking skeletons of these two programs electronically available at "http://www.cs.uno.edu/~shengru/pub/metri_ada".

### 4.5 Token Ring Program

The above three examples have illustrated how our metrics measure programs. The program in this subsection demonstrates how our metrics provides feedback for a design. This is a program of mutual exclusion within a token ring. An entry call diagram of an Ada implementation based on the algorithm given in [Andrews91] is shown in Figure 4.1. Two tasks are used for each node in the ring.

There are 5 communication points in each Helper task and 3 in each Process task. Let $n$ be the number of nodes in the ring. In Version 1, FCL(i, Helper(i).Enter) = 5n+3(n-1), because the control flow starting from task Process(i) can not only reach to every Helper task but also every other Process task. By swapping the entry call and accept relationship, we obtain Version 2 whose entry call diagram is shown in Figure 4.2. The value of FCL(i, Helper(i).Enter) is reduced to 5n, observing that other process tasks are not reachable from task process $i$. In real useful programs, this improvement would be more important, since processes are typically more complex than the ring itself. Isolating the control flows of process tasks would reduce the effort in debugging. We have accomplished this by taking advantage of Ada tasking's feature that separates control flows from data flows. That is, an *entry call* statement can serve for both sending and receiving data through its parameters. So can an accept statement.

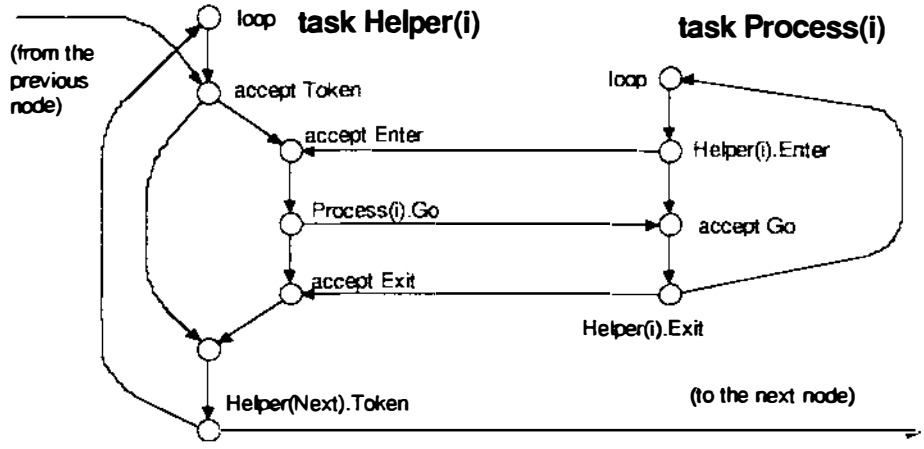



Fig. 4.1 Token Ring - Version 1

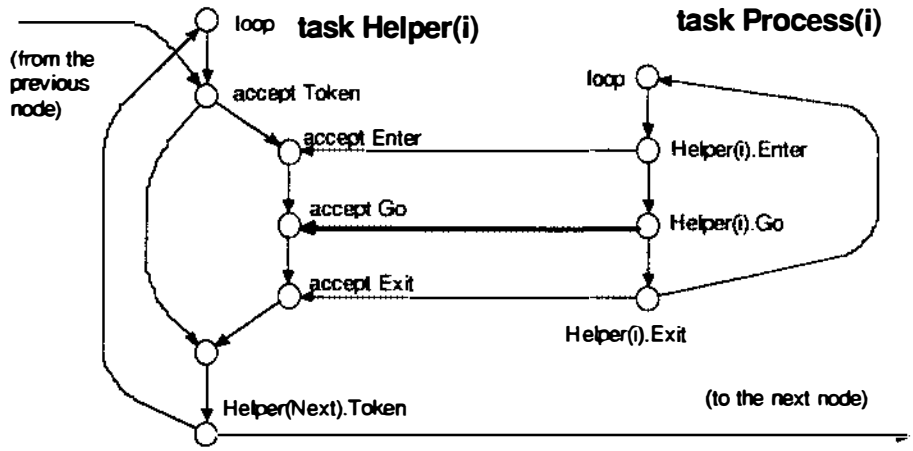



**Fig. 4.2 Token Ring - Version 2**

## 5. Conclusion

Our metrics can be applied in the design, implementation and maintenance stages of software development. One of the advantages of our metrics is that they can be applied to synchronization skeletons representing large concurrent systems at very early stages in design. Of course, our metrics can also be applied to final source code. Typically, metrics TC and TCC measure or predict the sheer work load in the implementation of different tasks, metrics FC and BC indicate the importance of program components, and FCL predicts the difficulty in tracing the execution of programs. Another advantage is the simplicity of the proposed metrics. Automation of the measurement is straightforward. Software engineers can easily understand what properties are measured.

As shown by the theoretical evaluation, our metrics identify single components as being less complex than when joined with other components and reflect the complexity imposed by the order of communication statements and interactions among them. The results of the theoretical evaluation and experimental calculations have led us to a number of hypotheses of using our metrics. TCC and PCC identify communication complexity due to local control flows within each task and can be used to estimate the design and coding effort of each task. FC and BC measure the range of task interaction caused by direct and indirect communication and are closely related to the workload in testing. FCL reflects the complexity of the causal chains in synchronization structures and is an indicator to point out the difficult part of a program in debugging.

Experiments conducted with five well-understood programs of various sizes and types have illustrated phenomena consistent with our hypotheses. However, the testing and refinement of these hypotheses will be a long term work. One of our goals is to establish some general threshold for our metrics as diagnostic indicators. However, it would be premature to do so before extensive experiments have been conducted. FCL has shown its capability in predicting difficulties of program components (tasks) on a relative basis. Our future work will include accumulating the measurements of the proposed metrics for various types of software systems as well as the actual development process. A set of reasonable thresholds will then be statistically established.

**Acknowledgment**

This work is partially supported by a 1994 UNO Science Research Award. The authors would like to thank George Brown for his implementation of the token-ring algorithm in Ada, and Dr. Sol

Shatz and Dr. George Avrunin for providing example programs. Anonymous reviewers have given us very constructive comments, which greatly helped us to improve this paper.

## References

[Adrews91] G.R. Andrews, Paradigms for Process Interaction in Distributed Programs, ACM Computing Surveys, 23(1): 49-89, March 1991.

[Byrne91] Byrne, W.E., Software Design Techniques for Large Ada Systems: Chapter 7, Digital Press, 1991.

[Cote88] V. P. Bourque, S. Oligny, and N. Rivard. Software Metrics: An Overview of Recent Results. Journal of Systems and Software, 8(2):121-131, March, 1988.

[Demerla92] S. Damerla and S. M. Shatz. Software Complexity and Ada Rendezvous: Metric Based on Nondeterminism. Journal of Systems and Software, 17: 119-127, February 1992.

[DeMillo87] R.A. DeMillo, W.M. McCracken, R.J. Martin and J.F. Passafiume. Software Testing and Evaluation. Benjamin/Cummings Pub., Menlo Park, CA, 1987.

[Gehani91] N. Gehani. Ada concurrent Programming (second edition). Silicon press, 1991.

[Gomaa94] H. Gomaa. Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley Pub., Reading, Mass, 1994.

[Halstead77] M. H. Halstead. Elements of Software Science. New York: Elseview North-Holland, 1977.

[Helmbold85] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," IEEE Software, 2(2), March, 1985, pp. 47-57.

[Levine93] D. L. Levine and R. N. Taylor. Metric-driven Reengineering for Static Concurrency Analysis. Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA '93), T. Ostraud and E. Weyuker, ed, Cambridge, Mass., pp. 28-30, June 1993.

[McCabe76] T. J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, SE-2(4):308-320, December 1976.

[McCabe94] T.J. MaCabe and A.H. Watson, "Software Complexity". CrossTalk - Journal of Defense Software Engineering, 7(12):5-9, Dec. 1994.

[Oviedo80] E. I. Oviedo. Control Flow, Data Flow and Program Complexity. Proceedings of the IEEE COMPSAC, Chicago, IL, November 1980, pp. 146-152.

[Perlis81] A. Perlis, F. Sayward, and M. Shaw (editors). Software Metrics. MIT Press, Cambridge, Mass., 1981.

[Shatz88] S. M. Shatz. Towards Metrics for Ada Tasking. IEEE Transactions on Software Engineering, 14(8):1122-1127, August 1988.

[SPC89] The Software Productivity Consortium. Ada quality and style: guidelines for professional programmers. New York: Van Nostrand Reinhold, 1989.

[Weyuker88] E. J. Weyuker. Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, 14(9):1357-1365, September 1988.

## Appendix

**Program 1: Readers-writers program**

```
task body READER_1 is
begin
    loop
        R_W.START_READ;
        X := S;
        R_W.END_READ;
    end loop;
end READER_1;
```

```
task body WRITER_1 is
begin
    loop
        R_W.START_WRITE;
        S := X;
        R_W.END_WRITE;
    end loop;
end WRITER_1;
```

```
task body R_W is
   NO_READERS: NATURAL := 0;
   WRITER_PRESENT: BOOLEAN := FALSE;
begin
   loop
      select
       when not WRITER_PRESENT and START_WRITE'COUNT = 0 =>
          accept START_READ;
          NO_READERS := NO_READERS + 1;
       or
          accept END_READ;
          NO_READERS := NO_READERS - 1;
       or
          when not WRITER_PRESENT and NO_READERS = 0 =>
             accept START_WRITE;
             WRITER_PRESENT := TRUE;
       or
          accept END_WRITE;
          WRITER_PRESENT := FALSE;
          for I in 1..START_READ'COUNT loop
             accept START_READ;
             NO_READERS := NO_READERS + 1;
          end loop;
      end select;
   end loop;
end R_W;
```

**Program 2: Gas Station Program (1 pump and 2 customers)***

```
task body PUMP is
 begin
  loop
   accept ACTIVATE;
   select
    accept C1_START_PUMPING;
   or
    accept C2_START_PUMPING;
   end select;
   select
    accept C1_FINISH_PUMPING;
    OPERATOR.C1_CHARGE;
   or
    accept C2_FINISH_PUMPING;
    OPERATOR.C2_CHARGE;
   end select;
  end loop;
 end PUMP;
```

```
task body CUSTOMER_1 is
 begin
  loop
   OPERATOR.C1_PREPAY;
   PUMP.C1_START_PUMPING;
   PUMP.C1_FINISH_PUMPING;
   accept CHANGE1;
  end loop;
 end CUSTOMER_1;

task body CUSTOMER_2 is
 begin
  loop
   OPERATOR.C2_PREPAY;
   PUMP.C2_START_PUMPING;
   PUMP.C2_FINISH_PUMPING;
   accept CHANGE2;
  end loop;
 end CUSTOMER_2;
```

* This program was provided by G.S. Avrunin at University of Massachusetts

```
task body OPERATOR is
 C1_IN, C2_IN: BOOLEAN := false;
begin
  loop
   select
    when (not C1_IN) =>
     accept C1_PREPAY;
     C1_IN := true;
     if not C2_IN then
      PUMP.ACTIVATE;              --activate the pump for c1
     end if;
   or
    when (not C2_IN) =>
     accept C2_PREPAY;
     C2_IN := true;
     if not C1_IN then
      PUMP.ACTIVATE;              --activate the pump for c2
     end if;
  or
    when C1_IN =>
     accept C1_CHARGE;
     if C2_IN then
      PUMP.ACTIVATE;              -- activate the pump for c2
     end if;
     CUSTOMER_1.CHANGE1;
     C1_IN := false;
  or
    when C2_IN =>
     accept C2_CHARGE;
     if  C1_IN then
      PUMP.ACTIVATE;              -- activate the pump for c1
     end if;
     CUSTOMER_2.CHANGE2;
     C2_IN := false;
  end select;
 end loop;
        end OPERATOR;
```

# Analysis Errors Estimation in a Large MIS: Two Empirical Case Studies.

**David Gefen**
Department of Computer Information Systems
College of Business Administration
Georgia State University
P.O. Box 4015
Atlanta, GA, 30302-4015

Tel. (404) 651-3880 (W)
(404 634-2739 (H)
E-Mail: CISDGE@SMTPGWY2.GSU.EDU

## ABSTRACT

Analysis and requirements errors discovered during testing are often the hardest to correct, especially in large MIS. Data from two sub-systems developed with IBM's CSP containing 124 and 62 KLOC, suggest that Software Science can be expanded and applied *also* to the estimation of *requirements* and *analysis* errors. This research proposes a much simpler metric to identify application susceptible to changes in requirements and analysis. The new metric is a variation on the theme of entropy and does not distinguish between operands and operators. The new metrics adjusted $R^2$ are consistently higher than those obtained with Software Science or lines-of-code.

Based on these findings, the feasibility of incorporating also *requirements* and *analysis* related error estimation into software and cost assessment metrics is discussed. The potential to effectively identify the applications more susceptible to such errors, could present significant advantages to management and quality assurance, especially as it may be done at an early stage of testing and development. Further research is this vein is called for.

**Keywords:** Software Metrics, Software Science, Entropy, Software errors

**Biographical sketch:** Mr. Gefen has a M.Sc. from Tel-Aviv University and is currently enrolled in the Ph.D. program in CIS at Georgia State University. Mr. Gefen has spent the last 14 years in software development dealing with several large scale MIS. During this time he held various positions including those of chief programmer, system analyst and project manager.

## 1. Introduction

Building new software systems is a costly endeavor where a significant proportion of the overall cost is spent on correcting software errors. Estimates put the percentage of cost devoted to errors at between 35% and 50% of the programmers' time (Dunn, 1984) and more than 50% of the total development cost (Beizer, 1984; Capers, 1986). By and large, this is regarded as the most expensive activity in software development (Capers, 1986) and is theorized to be directly related to the effort and time devoted to software development (Halstead, 1977). Consequently, many software metrics attempt to estimate the number of software errors and incorporate them into cost assessment models.

The importance of assessing software error distribution, in general, has been extensively discussed in the literature. Shooman (1983) and Gotlieb (1985), for example, present its importance to managerial decisions concerning methodologies, design and to effort and applicability estimation. Dunn and Ullman (1982), on the other hand, focus on quality and present its importance for the creation of thresholds for software reliability and performance. Still other arguments are presented by Capers (1986) concerning the importance of *measurements* in assessing a technology. Beizer (1984) sums up the issue by simply stating that the ability to apply quantitative measurements is a characteristic of the transition from craftsmanship to science.

Many software error models, such as McCabe's metrics and Halstead's software science, focus on *programming* related errors (Beizer, 1984), and, accordingly focus on the written *code*. However, software also suffer from errors in the *analysis* process that precedes writing the code and from *performance* problems that plague it. Such a distinction between *programming* and other types of errors was presented by Swanson (1976) in the context of software maintenance and have been widely quoted, for example by Lin (1988), Arthur (1988) and Chong Hok Yuen (1989). Swanson (1976) distinguished between *Corrective, Adaptive* and *Perfective* software changes. The corrective activity deals with fixing *programming* errors. The perfective activity improve the existing software without adding new features to it, for example by improving its *performance*. The adaptive activity modifies the software to answer new needs of the organization and to adapt to a changing environment. The adaptive activity during maintenance is similar to the changes and corrections in the *requirement*s and the *analysis*, that occur during testing. Where, *requirement* errors relate to mistakes made by the client, and *analysis* to the analyst. This distinction is similar to that presented by Beizer (1990) concerning software testing, on which this research is based and which are presented in Table 1.

Of the software related errors, *requirements* and *analysis* errors are probably the most costly type to correct (Brooks, 1987). Yet, these have not received due attention in traditional software metrics, even though, according to some statistics, these account for almost 25% of the total software errors (Beizer, 1990). The ability to estimate the number of these errors could contribute to better cost and time assessment of software projects, probably even more than programming error estimation does, because of the prominent role analysis mistakes have in increasing software cost. Furthermore, the effect of analysis related errors is of growing contemporary importance given the current trend to develop more complex software. A trend that has been tentatively shown to be associated with a relative increase in analysis related errors (Gefen and Ariav, 1992).

The importance of errors related to *requirements* and *analysis* is especially significant in large MIS. Such systems usually contain many related and complex applications, to such an extent that analysts and clients often find it hard to grasp the entire complexity of the system until it is

functional. Consequently, many problems related to the requirements are not located in the traditional "walk-through" checks during the analysis and development phases. These problems often arise only when many applications are combined and tested together at the later stages of testing. At this late stage in software development, changing the specifications can prove to be a very costly endeavor. The ability to locate requirement and analysis error prone applications can, therefore, be a prime factor in cost and quality assessment, especially in large MIS.

The motivation for this research began during the development of a one such very large MIS dealing with logistics. In this set of software development projects, it became clear at a very early stage that *requirements* given by the client were not perfect, as too, were the system analysts' understanding and subsequent *analysis*. Since requirements and analyses related problems can be more readily negotiated with the client paying for the software, and since these errors and additions are usually more expensive to correct after the software has been delivered, the need for an early identification of these applications during testing became important. Specifically, a metric was sought that could identify applications with a high probability of requirements and analysis related errors. Subsequent discussions with many other project managers and systems analysts revealed that this need was a common one.

The case studies described apply to two of a number of sub-systems developed in 1990 and 1991, in which the researcher took part first as a systems analyst and later as the software maintenance manager. An overriding problem that was experienced during testing and deployment was that the errors, and especially the analysis related ones, were not evenly distributed among the applications. In fact, most of the problems resided in a small number of applications. Most of the systems' analysts had no difficulty identifying these application, based on their subjective assessment of the *complexity* of requirements and application type. However, there was no objective measure to apply. Hence, this research set out to find an applicable and simple metric that would be applicable to errors pertaining to *analysis* and to *requirements*.

The initial step taken was to examine existing metrics. However, despite the importance of errors relating to requirements and analysis, software metrics have focused mainly on programming errors. An exception to this tendency is Halstead's (1977) software science metric. Halstead's book actually relates explicitly also to the realm of *English prose*. Based on the observation that requirements are written and conveyed in natural languages, this research first focused on software science and later sought to simplify the metric. The following sections will describe the metric applied, followed by sections on the research questions, research method, results, metric evaluation and discussion.

## 2. The Software Science and Entropy metrics:

Halstead's (1977) theory shows that a direct derivative of program length is the effort needed to create it and the number of errors at delivery that it will contain. Halstead's metric focuses on *programming* related errors only. Nonetheless, the applicability of Halstead's metric to other types of software related human activities is suggested in software science itself, where Halstead uses software science in the context of natural languages too. There are two elements of software science that this research focused on. The first is the ability to estimate the length of the code before it is written on account of the number of unique operands and operators. The second is the ability to estimate the number of delivered programming errors on account code length.

This latter element is explained in terms of human thought limitations rather than computing language limitations.

Halstead's metric has been extensively researched in assembler and various 3GLs showing very high $R^2$ (coefficients of multiple determination), ranging above 90% in estimating both code length and the number of programming errors. Some of these results are presented by (Fitzsimmons and Love, 1978; Ottenstein, 1979; Dunn and Ullman, 1982; Christensen, Fitsos and Smith, 1981). Among software metrics, the ability to estimate the length of a software program is *unique to Halstead's metrics* (Beizer, 1984). This ability has great potential, because as Beizer (1984) and Shen et. al. (1985) show, the number of unique operands and operators can be reasonably well estimated at early stages of software development. As a result, the number of programming errors can be estimated on account of code length estimation, which in turn, can be estimated once the number of *unique* operands and operators is approximately known, i.e., even prior to writing the code.

The applicability of code length estimation is not limited to software. Indeed, Halstead (1977) showed its applicability to English too. The total length of a text (number of words) can be estimated according to the number of *unique* operands (such as nouns) and operators (such as verbs) in contains. This is done based on the assumption of parsimonious writing. This ability is not surprising considering that, mathematically, Halstead's method is closely related to Shannon's entropy theory and can also be directly derived from Zipf's Law in linguistics (Shooman, 1983). Zipf's Law, states that the length of a text in a natural language is closely related to the number of unique words within it. Zipf showed this with a variety of natural language: English, Mandarin and Latin (Shooman, 1983). The similarity of natural languages and their length estimation according to Zipf's Law, on the one hand, and the algorithm length, on the other, was also shown by Zweben (1977).

Based on its similarity to natural languages, the research first attempted to apply this metric also to non-*programming* related errors. In addition, because Zipf did not distinguish between operands and operators, a new metric, which is a *variation of entropy*, was examined (for simplicity, the metric will be called the proposed entropy metric in subsequent references)[i]. Entropy is closely related to Halstead's metrics. However, it does not distinguish between operands and operators. This, in effect, means that the metric can be applied to the code without a comprehensive and indepth analysis to distinguish between operands and operators. Thus achieving a much *simpler and cheaper metric*. The entropy metric does not to distinguish between operands and operators because both are assumed to be manifestations of the same mental processes that relates to requirements' comprehension by the client (*requirements errors*) and the analyst *(analysis errors*). This is in contrast to Halstead's metrics, where operands and

---

[i] The original version of the metric was the actual entropy:

$N * \text{Log}_2 (n)$ by application_type

where "n" is the number of unique code tokens (operands and operators) and "N" the total number of code tokens. The above metric is, in fact, Halstead's Volume without distinguishing between operands and operators, and weighting for application type. However, the formula presented in this paper resulted in very similar adjusted $R^2$ and so was elected on account of nomological validity, i.e., showing similar predictive validity with less constructs.

operators are software related attributes fulfilling different objectives. Our metric, a variation on *entropy*, is defined as follows:

Entropy = N * $Log_2$ (N) * application_type

Where, "N" represents the total number of elements in the code, both operands and operators, and "application_type" represents the type of application, for example: queries, reports, on-line update screens, and the like. In building this metric two assumption were made. First, it was assumed that the entropy quantity will be highly correlated with the number of errors pertaining to requirements and analysis. This correlation is hypothesized based on the observation that complex applications usually have more changes and correction related to requirements and analysis than other applications do. The second assumption was that different application types will show a different susceptibility to requirement errors. For example, applications that deal with on-line updates are assumed to be more likely to have incomplete requirements than batch reports. Both assumptions were made, among other things, based on discussions with project participants.

## 3. Research Questions and Hypotheses:

The first research question examined whether the *estimated* program length estimates the *actual* program length. This construct is an elementary issue in Halstead's Software Science and all the subsequent program properties are closely related to it. Hence, establishing that the length can be effectively estimated is a *sine qua none* of this research.

**RQ1:** *How well can the program length be estimated?*

Based upon past research that has extensively examined Halstead's metrics in many computing environments and found it applicable, exhibiting very high $R^2$s in estimating actual program length (e.g.: Beizer, 1984; Fitzsimmons and Love, 1978), measured as the total number elements in it, we hypothesize that in a 4GL too, such as CSP, it will be applicable. This hypothesis is also supported by the reasoning behind Software Science as presented by Halstead (1977). This reasoning is independent of the programming environment and language, and even explicitly accounts for the difference in the *levels* of different languages. The first hypothesis was:

**H1a:** *Software science will predict actual program length.*

Halstead's length metrics are mathematically the summation of the operand's entropy and the operator's entropy. However, Shannon's information entropy concept does not classify the information items. Nor does Zipf's law. We therefore hypothesized that the number of *unique* elements would significantly estimate program length as measured according to the *total* number of elements that construct it. This assumption is very similar to Halstead's, with the difference that we do not distinguish between operands and operators in the estimation of parsimonious code length:

**H1b:** *Program length can be significantly predicted also without distinguishing between operands and operators.*

Assuming that program length estimation is indeed applicable to 4GLs too, as posited by the above hypotheses, it was hypothesized that error rates would also be applicable. These errors are defined literally as the *"number of delivered errors"* (Halstead, 1977, p.85). Halstead does not distinguish among bug types, but it seems clear that these refer to errors related to writing the software itself. These, however, are not the only type of error that plague software. Beizer (1990), for example, actually gives 763 distinctly different bug types combined into 9 categories. The second research question deals with the applicability of the Software Science to the estimation of all the different types of software related errors.

**RQ2:** *How well can the number of delivered errors, by the type of error, be predicted?*

Halstead (1977) reasoned the error estimation formula based on the related human thought process. The following hypothesis follows this line of reasoning and posits that errors that result directly from the human thought process will be significantly related to the program volume, as defined by Halstead, and will be estimated with high $R^2$s. These include errors that relate to requirements' *analysis*, to design and to *programming* errors, all of which deal directly with writing down thoughts, whether in a natural language or a programming one. *Performance* problems, on the other hand, relate more to data base related adjustments than they do to writing the code. Hence, performance problems are hypothesized not to be related to the program volume.

**H2a:** *The number of errors, both total and according to error type, can be significantly estimated applying software science, except for performance problems.*

Halstead assumed that the number of errors in a code would depend upon its volume, *i.e.*, the *minimal* length in bits of all the individual program elements. Volume is, thus, dependent upon code vocabulary, which suggests that programmers think in a programming language, rather than in a natural language, and that language directly affects thought. The former assumption seems questionable, given that programmers can easily learn new programming languages effectively reusing their existing knowledge. The implied assumption in software science corresponds to the *Whorfian Hypothesis*, that language affects human thought and that consequently people using different languages will think differently because of the different emphases placed by different languages. Apparently, this controversial hypothesis is not widely supported by research (Anderson, 1985). Consequently, it is hypothesized that the number of errors would depend mainly on information content, i.e., the *entropy* metric, and not on vocabulary.

This assumption can be tentatively supported by the assertion that requirements' related problems are caused in part by problems in human communications. As far back as 1978, Boland (1978) highlighted the many problems that originate in imperfect user and analyst communications. These problems were still highlighted as one of the major problems in IS development by Salaway (1987). To our knowledge and experience, this still is the major problem in IS development. If indeed communication is a prime cause of requirement related errors, then a communication content based metric, such as entropy, could be an appropriate estimator too. Thus, the following hypothesis states that:

**H2b:** *The number of errors, both total and according to error type, can be significantly estimated applying the proposed entropy metric, except for performance problems.*

## 4. Research Method:

Two extensive case studies were conducted in an preliminary attempt to show the effectiveness of Halstead's metric for the estimation of *requirements* and *analysis* errors, in both the original and the proposed simplified versions. These focused on examining whether this software metric that, in the past, has been shown to effectively estimate *programming* related errors could be applied to the estimation of *requirements* and *analysis* related software errors too, and whether the metric is applicable to contemporary development methods, such as 4GL.

The software was developed for a very large organization where it was successfully implemented to serve the organization's logistics needs and served as a strategic MIS through which policies were implemented. The project was managed from its initial development stages in accordance with the ISO9000 quality control standards. Due to its complexity and size, the project was regarded as a cornerstone project and received thorough examination by both the organization it was implemented in and the software vendor. The following sections will describe the project and its environment, followed by the way the software was tested and how the error data was collected.

The software inspected was written in CSP (Cross System Product) version 3.2.2, a state-of-the-art MIS centered 4GL from IBM. CSP was run in CICS and TSO environments and used DB2 version 2.2. All these tools are in the forefront of IBM's mainframe software and are key elements in IBM's SAA. The software itself was developed by a leading software vendor, that had developed similar software in the past, and was conducted in close cooperation with IBM. In this project, contemporary software development practices were used, such as extensive use of an *application generator*, of *code hiding* techniques and of a reusable *Software Architecture* method of creating new applications out of sample frameworks. Application generators and Software Architecture are both contemporary leading techniques in applying software reuse (Krueger, 1992).

The project examined was a 250 KLOC of 4GL containing at the installation stage 207 applications, divided into 4 sub-systems developed by separate teams. Two sub-systems were examined for this research, containing 102 applications, 79 in one sub-system and 23 in the other, totalling 104 and 62 KLOC respectively. Two additional sub-systems were not included in the data collection. One because it was lead by the researcher and the other because it was not completed on time. The investigation was supplemented with ongoing informal interviews during the 18 months of data collection with programmers, analysts, DBAs, system programmers, testers, managers and users.

The software was managed in three separate environments: development , testing and operational. The software was developed in the *development* environment, where it was also tested by the team leader. After receiving the team leader approval and correcting all the open SPRs, the application was passed to the *testing* environment were the testing team examined it. Following their approval, the application was passed to the *operational* environment where it was tested by the client for correctness as well as by the DBAs for performance. Finally, the software was tested as a whole in a trial implementation that lasted four months. The error reports were classified into problems relating to *specifications* (requirements and analysis), *programming* or

*performance*. This classification was done due to the unique nature of each of these error types and because of the way the client was charged. The need to charge separately for software modifications due to errors in requirements was agreed upon at an early stage of the software development when it became obvious to all parties involved that getting the requirements absolutely accurate and comprehensive before the system was built was not a reasonable demand.

During the various testing stages, every error and modification in the software were meticulously recorded. Every upgrade was pedantically recorded and every Software Problem Report (SPR) registered and classified. All the SPRs concerning the examined sub-systems were studied. The SPR's structure and filling regulations are similar to those of other researched projects, as described by Shooman (1983). Each SPR, when filled by the tester, contained the application name, date and a detailed description of the error. The SPR was then passed to the team leader who passed it on to the programmer. After correcting the software, the programmer added a detailed description of the modification performed in the software. All the SPRs were collected and later categorized by the team leaders according to error type (requirements, programming or performance) and error severity. The detailed classification into Beizer's (1990) error types was done manually at a later stage. Table 1 presents the error classification used.

The data collection method of this research was planned so that it would not influence the research validity. None of the participants knew of the ongoing research, except for top management who gave their approval to collect and use the data for subsequent research. The interviews were all non-formal discussions that were conducted as an ongoing activity between team members. These included the programmers, team leaders, testers and the client's industrial engineers. The interviews also served as an important aid in building the hypotheses and research questions. During the data collection phase of the research, the researcher was in charge of a third sub-system, that was not included in this research. This position gave the researcher an insider's access to all the project managers, clients, system analysts and programmers on a routine basis.

In addition, the characteristics of each application, such as lines-of-code, were collected by a specially written program that examined the CSP application printouts. This program presented, for each CSP application, the number of unique and total operands and operators, as well as other details. These characteristics were collected for each application after it was debugged by its programmer and passed to the team leader for initial testing. The software characteristics presented by this program were manually verified in a sample of programs.

The SPR collection period began with the initial coding and ended at installation. Thus it covered the entire development and testing periods. The SPR collection ended with the successful implementation of the system, indicating that the software was reasonably stabilized, hence covering most of the errors in it. The SPRs of each CSP application were matched with its software characteristics for the statistical analysis.

To show the reliability of the metrics, the research examined two independent sub-systems. Each sub-system dealt with different topics, was developed by different teams, had different clients and users, and had its own databases and applications. Thus, two independent assessments of the examined metrics could be complied and compared. The two sub-system, though independent from each other, were taken from the same project to increase internal validity, i.e., the isolation of cause and effect from other possible factors (Cook and Campbell, 1979). Comparing similar yet independent case studies, often referred to as literal replication (Yin, 1984), is a recommended method for increasing the reliability of the results (Benbasat et. al. 1987).

| | |
|---|---|
| 1) Requirements | These relate to inconsistent, ambiguous and incomplete requirements as well as to changes in the requirements. |
| 2) Functionality as implemented | These relate to the correctness and completeness of requirements as understood. It also relates to domain bugs, user messages and diagnostics and to exception condition handling. These errors are related to as *analysis errors* in the examined sub-systems. |
| 3) Structural | These relate to errors in algorithms, logic, control, initialization etc. |
| 4) Data | These are errors in data definition, access and handling. |
| 5) Implementation coding | Errors in coding, standards and documentation. |
| 6) Integration | Errors concerning interfaces, parameters and timing of the integration between software components. |
| 7) System (Performance) | These relate to operating system calls, locks and resource management and issues of performance. In the context of this research the term is limited to performance problems only. |
| 8) Testing | Problems in the definition and execution of tests. This category was not examined in this research. |
| 9) Others | This category was not examined in this research. |

**Table 1: Software Error Types According to Beizer (1990)**

In the sub-systems analyzed, there were 6 types of applications: on-line update applications, queries, batch applications, reports, CICS background applications to print forms, and applications to order reports. Each application, by definition, belonged to one of the above categories and was developed by modifying a skeleton application designed specifically for that type of application.

## 5. Results:

The analysis of the SPRs was performed using SPSS version 6.0. The results are presented in this section in the order of the hypotheses.

**H1a:** *Software science will predict actual program length.*

The results of this examination, applying linear regression, are presented in Table 2. Each application was analyzed as one data point in its sub-system. The results presented in Table 2 show very high $R^2$ ,these, however, are not exceptional to these sub-systems. Similar results, ranging from 90% to 99%, are presented by Fitzsimmons and Love (1978) and by Christensen, Fitsos and Smith (1981). The results clearly show that software sciences' estimate of length applies to CSP in this project too and *confirms the hypothesis.*

| | Sub-system 1 | Sub-system 2 |
|---|---|---|
| adjusted $R^2$ | .96 | .98 |
| F value (P value) | 1704.17 (.0000) | 1108.48 (.0000) |
| B (regression coefficient) | .8317 | .9569 |
| standard error of B | .0202 | .0287 |
| number of applications | 79 | 23 |
| total KLOC in sub-system | 124 | 62 |

**Table 2: Actual Program Length As a Function of Estimated Program Length.**

Even though the sub-systems are quite different from each other, the dependence of the actual length on the estimated length is very highly significant in both, indicating the high reliability of the estimation. Furthermore, in both sub-systems, the estimated length is close to the actual length, and the regression coefficient has very small standard errors. The difference between the two sub-systems is probably due to their different circumstances, beyond the obvious difference of the teams involved. In sub-system 1 it was discovered, half way through development, that an entire topic had been overlooked. As a result, many unforeseen additions and changes had to be made during development, which may account for the slightly lower $R^2$ in this sub-system.

**H1b:** *Program length can be significantly predicted also without distinguishing between operands and operators.*

The results of this examination, applying linear regression, are presented in Table 3. The metric examined in this hypothesis is equivalent to Halstead's metric in the previous hypothesis except that it does not distinguish between operands and operators.

| | Sub-system 1 | Sub-system 2 |
|---|---|---|
| adjusted $R^2$ | .93 | .96 |
| F value (P value) | 998.25 (.0000) | 608.65 (.0000) |
| B (regression coefficient) | 9.4068 | 11.7773 |
| standard error of B | .2977 | .4774 |

**Table 3: Actual Program Length As a Function of the Entropy of Unique elements.**

Estimating program length according to the entropy of the unique elements that construct it, also shows very high $R^2$, and very significant F values. *The results confirm the hypothesis.* The ability to measure application length according to the total number of elements it contains is considerably easier than applying software science's length because no classification to operands and operators is needed. Furthermore, this metric combines both simplicity of measurement, it is in fact almost as simple as KLOC (kilo-lines-of-code), and a sound mathematical reasoning based on Halstead's parsimonious code and Shannon's entropy. Thus, while entropy, like software science length, avoids many problematic measurement issues of KLOC, it nonetheless, can be

measured without the complexity of measurement associated in distinguishing between operands and operators.

The second research question dealt with estimating the number of delivered errors. This was analyzed for each error type category described by Beizer (1990) and presented in Table 1.

**H2a:** *The number of errors, both total and according to error type, can be significantly estimated applying Software Science, except for performance problems.*

Table 4 sums up the linear regression and ANOVA results of predicting the number of errors of each type according to Software Science's bug estimation (third column). The linear regression was applied with no intercept, based on the assumption that no code has no errors. This assumption was supported when the linear regression models were run with intercepts. In these, the intercepts was almost zero and all had very insignificant T values.

As can be seen from Table 4, performance problems stand out as the only type that shows either insignificant F values or very low $R^2$, in both sub-systems. Otherwise, the other error types show, overall, extremely low P values, implying that *the hypothesis is supported* by the data. The generally higher $R^2$ of the requirements and functionality errors in comparison with the other error types, shows that not only can Halstead's metrics be applied to analysis related errors too, but that the percent of variance they explain of these error types is larger than that explained for programming related activities (structural, data, implementation coding and integration). The large difference in the $R^2$ of *requirements* and *analysis* between the two sub-systems is probably due to their different circumstances, described above.

**H2b:** *The number of errors, both total and according to error type, can be significantly estimated applying the proposed entropy metric, except for performance problems.*

Table 4 (first column) sums up the linear regression and ANOVA results of predicting the number of errors of each type according to entropy and application type. The linear regression was applied with no intercept. Here too, the intercept, when applied, showed very insignificant T values and was almost zero.

Table 4 *confirms the hypothesis.* Furthermore, overall, the proposed entropy metric gives estimations that have higher adjusted $R^2$ values and F values and lower associated P values than Software Science. It can further be seem from Table 4 that the entropy metric is a superior estimator than lines of code are, in both adjusted $R^2$ values and P values. On average, the adjusted $R^2$ of the proposed entropy metric is more that *23% larger* for requirements and functionality errors than that obtained with KLOC. The high adjusted $R^2$ relating to the lines of code in this project can be explained as the result of programming procedures standards enforced by the chief programmer team. As a consequence of these, the applications were developed out of skeleton sample applications, and contained single line, non-embedded commands.

The original need to develop this metric was to identify applications with many analysis related errors. This objective was achieved. The average adjusted $R^2$ of the requirements and functionality errors is 68%, suggesting that the metric is, at least in the case of the examined sub-systems where it was applied, a good and useful estimator of potentially error prone software.

Another way to examine the hypothesis is to test whether the metric can identify the applications that have a higher rate of requirements and analysis errors. To perform this test a

Discriminate Analysis was run and its ability to correctly classify the top 5% of applications with such errors was examined. In both sub-systems the estimation was significantly better than by chance alone. Wilks' Lambda[ii] was .4644 (P-value = .0000) in sub-system 1 and .6418 (P-value = .0119) in sub-system 2. The hit ratio, i.e., the percent of applications correctly classified, was 96% in sub-system 1 (3 applications were incorrectly classified) and 87% in sub-system 2 (3 applications mis-classified). Of the 6 applications that were incorrectly classified, only 1 was a top 5% that was not identified. Simply stated, the proposed metric did identify the applications more susceptible to requirements and analysis errors.

| Sub-system | | Entropy Metric adj. $R^2$ F value (P value) | | Lines of code adj. $R^2$ F value ( P value) | | Software Science adj. $R^2$ F value( P value) | |
|---|---|---|---|---|---|---|---|
| Requirements | 1 | .58 | 56.19(**.0000**) | .42 | 57.83(**.0000**) | .44 | 62.28(**.0000**) |
| | 2 | .73 | 32.37(**.0000**) | .73 | 59.64(**.0000**) | .71 | 57.31(**.0000**) |
| Functionality | 1 | .68 | 84.62(**.0000**) | .54 | 93.62(**.0000**) | .58 | 109.58(**.0000**) |
| (Analysis) | 2 | .73 | 32.41(**.0000**) | .57 | 30.72(**.0000**) | .73 | 62.99(**.0000**) |
| Structural | 1 | .26 | 14.86(**.0000**) | .21 | 21.70(**.0000**) | .23 | 24.50(**.0000**) |
| | 2 | .46 | 10.96(**.0005**) | .14 | 4.44(**.0472**) | .47 | 21.39(**.0001**) |
| Data | 1 | .23 | 13.11(**.0000**) | .13 | 12.78(**.0006**) | .23 | 25.13(**.0000**) |
| | 2 | .70 | 27.84(**.0000**) | .47 | 0.04(**.0002**) | .71 | 56.72(**.0000**) |
| Implementation | 1 | .60 | 59.72(**.0000**) | .46 | 68.32(**.0000**) | .51 | 81.94(**.0000**) |
| coding | 2 | .55 | 14.83(**.0001**) | .01 | 1.29(.2687) | .43 | 18.53(**.0003**) |
| Integration | 1 | .29 | 17.00(**.0000**) | .18 | 18.38(**.0001**) | .19 | 19.21(**.0000**) |
| | 2 | .55 | 15.30(**.0001**) | .44 | 18.04(**.0004**) | .53 | 27.18(**.0000**) |
| Performance | 1 | .06 | 3.54(**.0340**) | .02 | 2.47(.1205) | .07 | 7.25(**.0087**) |
| | 2 | .00 | 0.67(.52) | .00 | .09(.7642) | .02 | 1.42(.2457) |
| All SPR reports | 1 | .61 | 61.95(**.0000**) | .55 | 96.54(**.0000**) | .58 | 111.93(**.0000**) |
| | 2 | .84 | 122.28(**.0000**) | .71 | 53.67(**.0000**) | .84 | 112.45(**.0000**) |

**Table 4: Linear Regression and ANOVA Results of the Relation Between Entropy, LOC, Software Science and the Number of Errors by Type.**

## 6. Metric Evaluation:

Creating a single metric that captures all the diversity and richness of software complexity is impossible (Fenton, 1994). Nonetheless, there is room for metrics of specific complexity attributes that can be used for assessment or for prediction (Fenton, 1994). Such metrics need to show a clear interpretability of the results (Fenton, 1994), be general and relate to the perspective of the metric users (Schneidewind, 1992).

[ii] Wilks' lambda is used in discriminant analysis in a manner equivalent to the F statistic in ANOVA (Hair et. al. 1992)

The *entropy* metric proposed in this paper does all three. It addresses one aspect of software complexity: that which relates to the information content of requirements as measured by entropy. The *interpretability* of the metric is straightforward: the more words needed to describe a system, the more complex (and error prone) its requirements and analysis will be. The entropy metric is also a *general* metric as it does not relate to any specific software environment. Though the applicability of the proposed metric to other environments still needs to be shown by subsequent research. The metric also relates to the *users' perspective* in that it does not demand complicated measures and constructs, but, rather is intuitive. It is conceivable, that most people would agree that if a requirement can be made in a succinct manner, rather than by a lengthy and elaborate description, then it is more likely to be understood with less errors.

Once a metric has been created it needs to be validated. A methodology for validating such metrics has been proposed by Schneidewind (1992). This methodology contains six elements: association, consistency, discriminative power, tracking, predictability, and repeatability. *Association*, according to Schneidewind (1992) can be measured by the value of the $R^2$. The results section of this paper has shown that the $R^2$ adjusted for the degrees of freedom is high and associated with very significant linear regression coefficients. *Consistency* is measured by applying rank correlation. The results of Spearman's rank correlation are presented in Table 5, showing very significant rank correlation results. The *discriminative power* was measured using

| | Sub-system | Coefficient (P-value) |
|---|---|---|
| Requirements | 1 | .4516 (.0000) |
| | 2 | .7531(.0000) |
| Functionality (Analysis) | 1 | .5957(.0000) |
| | 2 | .8771(.0000) |

**Table 5: Results of Spearman's Rank Correlation.**

Discriminant Analysis to indicate whether a high versus low level of entropy (i.e., above or below average) can discriminate between applications' error rates. All the software error types had very significant Wilks' Lambda and associated F values, as indicated in Table 6. *Tracking* was not measured, as there was no evaluation over time. *Predictability* was shown by obtaining very significant results in linear regression. *Repeatability* has been shown to some extent by showing that two sub-systems had similar patterns of significance.

| | Wilks' Lambda | F (P-vlaue) |
|---|---|---|
| Requirements | .7811 | 28.02 (.0000) |
| Functionality (Analysis) | .5412 | 84.78 (.0000) |
| Structural | .8341 | 19.89 (.0000) |
| Data | .6872 | 45.52 (.0000) |
| Implementation coding | .6831 | 46.40 (.0000) |
| Integration | .7890 | 26.75 (.0000) |
| Performance | .9070 | 10.26 (.0018) |

**Table 6: Results of discriminant analysis.**

## 7. Discussion:

This research set out to propose an answer to an urgent need to identify requirements and analysis related error-prone software in a large MIS. This need is especially pertinent in large MIS due to the tendency of clients and analysts not to comprehend the entire system with all its complexity and connections until a preliminary product is displayed. This tendency leads to many costly changes in the *requirements* and reveals *analysis* mistakes at a late stage of the software development process. The ability to estimate these types of errors, and pin-point suspect applications at an early stage, is therefore, beneficial to both software cost assessment and software quality.

The applicability of the proposed *entropy* metric to answer this need was shown. The predictive ability of entropy is very high: an average adjusted $R^2$ of 68% combined with a hit ratio of 96% in one sub-system and 87% in the other. This suggests that the metric is, at least in the case of the examined sub-systems where it was applied, a good and useful tool for locating such applications. The incremental validity of the entropy metric over KLOC was demonstrated by an 23% improvement in the adjusted $R^2$. This result tentatively suggests that complexity is a better estimator of *requirements* and *analysis* errors than size is. Similar results about errors in general obtained *without distinguishing errors by their type*, are reported by Harrison (1992). Harrison, investigating large commercial applications, also found that while both complexity and size are good estimators of code related errors, nonetheless complexity is a better predictor.

The research showed that, as expected, Halstead's estimation of program length is highly significant. Not surprisingly, it accounted for more than 90% of the variance. The research also showed that a variety of programming errors are significantly dependant upon the program length. This result, too, was expected on account of Halstead's model and past research. The dependency of *requirements* and *analysis* related errors upon software science's program length was also shown. The latter hypothesized relationship is a unique contribution of this research. It shows that there is justification to apply Halstead's metrics beyond the original realm Halstead assigned for them, that of estimating *programmming* errors. Considering the higher cost of correcting requirements and analysis related errors in comparison with programming errors, this is an important finding and should be incorporated into project cost estimation models.

An important implication of these results is that error prone applications can be marked for the Quality Assurance testing team, not only for the written code itself, but also for the requirements and analysis upon which the code is based. To some extent, software testers and analysts can also be given an expected *range* of requirements errors that are likely to appear in a requirements document and a list of susceptible applications to focus on. Such an initial assessment could prove to be a valuable tool for analysts and software managers.

The reliability of the results was demonstrated. Both sub-systems, show a similar pattern of significant regressions and have very high adjusted $R^2$, higher than those of KLOC. However, the size of the data sets examined, and the case research method, do not enable an extensive analysis of whether this improvement over KLOC is a consistent pattern. The data do, however, suggest that even though KLOC can be applied also to estimate requirements and analysis errors, nonetheless, the proposed entropy metric is better. Identifying applications susceptible to changes in requirements and analysis using the proposed metric in addition to other possible metrics, such as KLOC, seems advisable.

Obviously further studies are needed to establish the generality of these results. Case studies are meant to indicate promising and interesting avenues and generalize to a theory. Case

studies are not meant to determine confidence intervals on the generality of parameters and metrics (Yin, 1994). Determining the interaction among the metrics, and consequently, which metric is better, would demand methods such as Structured Equation Modeling and Path Analysis. These demand a much larger sample size than was available. The results, so however, tentatively suggest the intriguing notion, that the information content of the requirements together with the type of the application is a prime factor affecting the number of requirement related errors. In other words, systems that can be defined by using a relatively small number of different keywords, are likely to have less errors. A result, that in hindsight, is not surprising.

The importance of this research goes beyond the estimation of the number of errors in written code. A unique attribute of Halstead's metrics is that the actual application size can be estimated from the number of unique operands and operators it uses. This ability was shown in this research too, for both projects, yielding not only significant results, but also an adjusted $R^2$ of above 90%. The consequences of this are vast, because, as Beizer (1984) points out, the number of unique operands and operators can be quite accurately assessed at a very early stage of the software development. Beizer (1984) actually quotes research putting the accuracy of this assessment at within 20% of the actual figures. Similar findings are reported by Shen et. al. (1985). If applicable to latge MIS projects too, then this ability could enable managers to identify the susceptible applications even before testing begins.

The ability to generalize these results to other programming languages and environments demands further research. On the one hand, the concept of entropy and its close relationship to error frequency has been widely investigated and verified in a wide variety of environments. Yet, on the other hand, making such a far reaching conclusion of generality on account of only two case studies, both of which were developed in the same language, is problematic. Consequentially, we feel that the use of entropy as presented in this research is a promising avenue of research, that builds upon established concepts, yet warrants further research before being widely applied.

## Bibliography:


Anderson J. R.. *Cognitive Psychology and Its Implications.* Second edition. W. H. Freeman and Company. NY 1985.

Arthur L. J., *Software Evolution. The Software Maintenance Challenge*, A Wiley-Inter science Publication John Wiley and Sons, New-York, 1988.

Boland, R. J., Jr., "The Process and Product of System Design". *Management Science*, Vol. (24:5), May 1978, pp. 887-898.

Beizer B., *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Company, NY, 1984.

Beizer B., *Software Systems Techniques.* Van Nostrand Reinhold Company, NY, 1990

Benbasat I., Goldstein D. K.. and Mead M., "The Case Research Strategy in Studies of Information Systems". *MIS Quarterly*. September 1987. pp.369-386.

Brooks F. "No Silver Bullet: Essence and Accidents of Software Engineering". *Computer.*April 1987 pp. 10-19.

Capers J., *Programming Productivity.*, McGraw Hill Book Company, NY, 1986.

Christensen K., Fitsos G. P. and Smith C. P.. "A Perspective on Software Science". *IBM Systems Journal.* Vol. 20(4). 1981. pp.372-387.

Chong Hok Yuen, C. K. S. C. H. "Differences in Types of Software Maintenance Work: An Empirical Study". *IEEE Proceedings. Conference on Software Maintenance*, IEEE Computer Society Press, 1989. pp.106-115.

Cook T.D. and Campbell D. T. *Quasi Experimental Design and Analysis Issues in Field Settings*, Houghton Mifflin Co. 1979.

Dunn R. H. *Software Defect Removal.* McGraw Hill Book Company. NY, 1984

Dunn R. H. and Ullman R., *Quality Assurance for Computer Software.* McGraw Hill Book Company. NY. 1982.

Fenton, N., "Software Measurement: A Necessary Scientific Basis". *IEEE Transactions on Software Engineering.* Vol. (20:3). March 1994. pp. 199-206.

Fitzsimmons A. and Love T. "A Review and Evaluation of Software Science". *ACM Computer Surveys* (10:1), 1978. pp. 3-18.

Gefen D. and Ariav G. "The affect of developing software with an application generator on software errors", *Proceeding of the sixth convention of the ADAMS institute of Tel-Aviv University,* 1992

Goldberg R. "Software Engineering: An Emerging Discipline". *IBM Systems Journal* Vol. (25:3/4), 1986. pp. 334-353.

Gotlieb C. C., *The Economics of Computers: Costs, Benefits, Policies and Strategies.* Prentice Hall, NJ. 1985.

Halstead M. H., *Elements of Software Science.* Elsevier Scientific Publishing Company, NY. 1977.

Hair, J. F. Jr., Anderson, R. E., Tatham, R. L. and Black, W. C., *Multivariate Data Analysis.* Third Edition. Macmillan Publishing Company. NY. 1992.

Harrison, W., "An Entropy-Based Measure of Software Complexity". *IEEE Transactions on Software Engineering.* Vol. SE-18(11), November 1992. pp. 1025-1029.

Krueger C. W., "Software Reuse". *ACM Computing Surveys* (24:2). June 1992. pp. 131-183.

Lin L. "Classifying Software Maintenance". *IEEE Proceedings. Conference on Software Maintenance,* IEEE Computer Society Press, Phoenix Arizona, 1988. pp. 241-247

Ottenstein L. M. "Quantitative Estimation of Debugging Requirements". *IEEE Transactions on Software Engineering.* Vol. SE-5(5), Sep. 1979. pp. 504-514.

Papoulis A. *Probability, Random Variables, and Stochastic Processes.* Third edition. McGraw Hill International Editions. Electrical & Electronic Engineering Series. 1991.

Salaway, G., "An Organizational Learning Approach to Information Systems Development," *MIS Quarterly*, Vol. (11:2), June 1987, pp. 245-265.

Schneidewind, N. F., "Methodology For Validating Software Metrics". *IEEE Transactions on Software Engineering.* Vol. (18:5). May 1992. pp. 410-421.

Shen Y. V., Yu T., Thebaut S. M., and Paulsen L.R. "Identifying Error Prone Software - An Empirical Study". *IEEE Transactions on Software Engineering.* Vol. SE-11(4), April 1985. pp. 317-323.

Shooman M. L., *Software Engineering*, McGraw-Hill, NY, 1983

Swanson E. B. "The Dimensions of Maintenance" *Proceedings of the 2nd international Conference on Software Engineering.* October 1976. pp. 492-497.

Yin, Robert, K., *Case Study Research Design and Methods.* Applied Social Research Methods Series Volume 5. Sage Publications. USA. 1984.

Zweben, S. H.,"A Study of the Physical Structure of Algorithms". *IEEE Transactions on Software Engineering.* Vol. SE-3(3), May 1977. pp. 250-258

# RELIABILITY OF MEDICAL SOFTWARE

Prepared for the 1995 Pacific Northwest Software Quality Conference

The use of computers in medicine is increasing rapidly, resulting in inevitable software failures. The nature and consequences of medical software failures are presented and categorized. The dilemma of the user, i.e., under what conditions it is appropriate to utilize medical software, and the degree of reliance that is justifiable once the software is utilized, is discussed and exemplified. Potential legal remedies to parties injured by failures of medical software are presented and critiqued. The Food and Drug Administration's efforts to regulate software in the health care industry are detailed, including the FDA's policy, mode of classification, perception of software development and approval requirements for medical software. The FDA's approach is applicable to the development of software in non-regulated environments.

Steven C. Schurr, Esq.
225 West Washington Street
Suite 2200
Chicago, IL 60606

Steven C. Schurr has law offices in Illinois and Indiana, concentrating in health care law. He acts as a Health Care Investigator for Schurr Health Care Corporation, a company promoting quality health care through consultation. Mr. Schurr began working with medical software in 1991, and became the recipient of the Academic Achievement Award for Computer Law at Loyola University of Chicago School of Law in 1992. He has sixteen years experience in quality assurance and regulatory affairs within the drug and device industry.

# RELIABILITY OF MEDICAL SOFTWARE

## I. INTRODUCTION

The current and potential uses of computers in medicine touch every aspect of health care. Errors in medical software are difficult to detect and may have tragic consequences. This paper discusses the current uses of computers in medicine, and the origin, nature and consequences of medical software failures, then considers the dilemma of the user, and examines the various legal theories available to an injured party. The paper also summarizes regulation of medical software by the Food and Drug Administration and discusses the FDA's efforts to reduce the frequency and adverse consequences of software malfunctions. Finally, two proposed plans to improve software accuracy are presented and compared to the FDA's regulatory requirements.

## II. USE OF COMPUTERS IN MEDICINE

Similar to any business or profession, computers are used to perform routine administrative functions.[1] In medicine, computers are particularly useful to facilitate research and education. Computer databases such as MEDLINE can be used by the physician to limit liability exposure for failure to make effective use of available information.[2]

Physicians estimate that the computerization of medical records will become more and more widespread,[3] raising legal issues such as admissibility as evidence, conformance to licensing and regulatory requirements, and security/privacy of data.[4] Computers are poised to play a major role in the assessment of quality of health care.[5] They are currently employed by the U.S. government to uncover Medicare and Medicaid fraud.

Computers can be used to automatically remind physicians when tests or immunizations are due or warranted.[6] Software programs are available that actively assist the physician in medical decision-making.[7] Hospital staffs routinely monitor patient status with the aid of computerized surveillance, analysis and response.[8] Computerized response can be passive, such as an alarm followed by human intervention, or active, such as direct alteration of patient therapy by "closed-loop feedback control."[9] The U.S. Food and Drug Administration regulates computer products designed to affect the diagnosis and treatment of patients.[10]

## III. NATURE AND CONSEQUENCES OF SOFTWARE FAILURES

Examination of recent medical device recalls[11] reported by the Food and Drug Administration demonstrate the nature and consequences of software errors in devices used to administer health care. From October, 1984 to August 1994, one-hundred sixty-seven (167) medical devices were recalled due to software or computer malfunctions.[12] Of these recalls, 5 represented a "potential for serious adverse consequences or death,"[13] 122 represented a potential for "temporary or medically reversible adverse health consequences,"[14] and 40 were "not likely to cause adverse health consequences."[15]

These software related recalls can be categorized into at least eleven different types, based upon the performance phenomena that resulted from the software defect or failure:

1. Input value dependent malfunctions. In these instances, the software fails to perform only when certain input values are entered. Often the input values prompting the failure are at the data extremes. For example, an infusion pump reported milligrams given as one-half the dose actually delivered when the drug concentration entered into the pump program was four milligrams per deciliter.[16] These types of malfunctions are difficult to discover by testing, because they are

prompted by only certain combinations of entered data.

2. Transposed data. Data is transposed when the software incorrectly places the data value in the incorrect field or data file. For example, a Magnetic Resonance (MR) System mismatched the imaging, patient, and other hospital information for one patient exam with the images from different MR exams.[17] These types of errors can result in misdiagnosis and inappropriate treatment.

3. Incorrect amount or timing of energy directly applied to the body by a computerized device. Considered to be the most serious, these types of malfunctions are exemplified by a linear accelerator radiation therapy machine that delivered massive radiation overdose to patients[18] and a ventilator that randomly provided extra breaths, either individually or in series, to the patient.[19]

4. Suppressed or deleted data. Sometimes, mere suppression of critical data can have disastrous results. Blood donor deferral information for 123 donors who had previously tested positive for the AIDs virus was deleted from computerized records, resulting in the distribution of blood and blood components collected from 13 of these donors. [20] In a less serious example, a computerized tomographic series X-ray system failed to display an exam when certain exam numbers were used on the scanner.[21]

5. Keyboard lock or shutdown. Medical devices may shutdown, making their use potentially dangerous or ineffective. A "simultaneous occurrence of relay component failure and invalid software conditions" caused an invasive monitor device to halt operation.[22] Similarly, a software defect has caused an anesthesia machine with a ventilator control system to fail.[23]

6. Malfunctioning alarm. Some medical devices contain alarms to warn the health care practitioner when clinical findings reach a level of concern; other devices contain alarms to warn when the device itself is not functioning properly. Warning alarms may be audio or visual. Alarms for high and low heart rates, low breath rate, high breath rate and high labored breathing index on a device designed to monitor respiratory and heart rate have been affected by software failures.[24] A critical care monitor lost its monitoring functioning output and its alarm functions due to loss of memory and data.[25]

7. Misinterpretation, misdiagnosis. If the health care practitioner is relying upon the computer to conduct calculations, and the results provided by the computer are erroneous, an incorrect diagnosis could result. For example, a software error in a diagnostic chemistry analyzer caused incorrect computation of curve statistics which could lead to misdiagnosis and error in treatment.[26]

8. Reversion to default settings. The software in some medical devices is programmed with default settings; these will be utilized in the absence of any contrary instructions by the user. If custom settings are entered, but the device unknowingly converts back to default settings before or during use, the device may malfunction. A ventilator used to deliver volumes of gas to patients dependent on artificial respiration was recalled because the ventilator settings could revert to factory default settings under certain conditions, and the ventilator would keep operating in the default mode.[27]

9. Data update error. The computer may erroneously save an older rather than a newer file. This may lead the health care practitioner to interpret data for a patient

that is actually for the preceding patient or for the same patient, but from a preceding examination. An electrocardiograph deleted new waveforms being entered into its system instead of old waveforms, whenever its directory of stored waveforms was full.[28]

10. Reversal of video image. A software/hardware change to the system of a scanner caused reversal of the video image.[29] If undetected, this could lead to misdiagnosis and treatment.

11. False impression that computer was adequately programmed. Memory software cartridges for use with pacemakers were recalled because the software could incorrectly confirm that the pacemaker was programmed, when, in fact, the pacemaker did not receive the programming transmission.[30]

The above-listed categories of medical device software malfunctions are not comprehensive and often overlap. They do, however, provide examples of the potential harm that can arise from a software malfunction in a medical device. A specific software malfunction, discovered in the field and resulting in a recall, could be difficult for the programmer to anticipate, because of a unique combination of events that expose the software defect. For example, a software system designed to analyze fetal growth rate from an ultrasound printed incorrect delivery data, overrode one report's results and input data onto another report, and incorrectly plotted the head circumference/abdominal circumference data of the first report, only if the second report was saved and then the first report was chosen to print.[31] In order to discover this mishap during product development, the manufacturer would have to test the device under this particular condition of use. Such an anticipatory testing process, called "hazard analysis," is a primary element in the FDA's regulation of computerized medical devices.[32]

## IV. DILEMMA OF THE USER

In health care, three legal questions arise for the potential user of a computerized system: 1.) when is utilization of a computer appropriate? 2.) to what extent is the user justified in relying upon the computer to perform a particular task? 3.) what recourse is available to the user or an injured third party if a computerized device malfunctions due to inadequate software design or failure?

### A. To Use or Not to Use

A computer program should be used in medical practice "if it improves the quality of care at an acceptable cost in time or money or if it maintains the existing standard of care at a reduced cost in time or money."[33] The mere existence of computers and computerized databases pose a legal threat to the health care practitioner, if the practitioner fails to use them under the appropriate circumstances. Courts in at least two cases have imposed liability for failure to utilize a new technology before its use becomes custom or widespread, especially when the technology is convenient, relatively inexpensive and construes significant benefit.[34]

### B. Is Reliance Justifiable?

Once a practitioner or an institution decides it is reasonable to use a computerized system or device, to what degree may the user rely upon the information provided by the computer, and to what extent may the user depend upon the computerized device to perform its intended function? In a health care setting, detrimental reliance on computer software can be life-threatening.[35] Because of the potential for serious harm, at least one court has imposed a high level of responsibility upon the user to monitor the computerized device, even though no input was required

from the user during the device's operation.[36]

In Schroeder v. Lester E. Cox Medical Center,[37] the Missouri Court of Appeals upheld an award of punitive damages and aggravating circumstances damages under a wrongful death statute against a pharmacy whose pharmacist failed to observe a computer driven compounder during the preparation of a surgical solution. The device in Schroeder is typical of many computerized medical devices because it performs, with the aid of a computer, a function that was performed manually before onset of the computer age. In this case, the device, a compounder, automatically mixed designated amounts of precursory solutions into a final solution of specific concentration.[38] This wrongful death action arose out of failure of the device to adequately mix a solution used in open heart surgery to protect the tissue of the stopped heart from damage. Because the resultant solution contained only 0.2% of the requisite amount of dextrose,[39] the cardiologist was unable to restart Irene Schroeder's heart after surgery.[40]

The compounder in question operated by automatically pumping premixed solutions of standard concentrations from bottles or bags through a load cell into a bag, thus creating a solution of specific concentration. The pharmacist used a control panel to program the compounder with instructions for pumping and mixing the necessary ingredients. First, she entered the specific gravity of each particular solution, then the volume desired. After each entry, the entered value appeared on the appropriate place on the control panel. After all entries were completed, the pharmacist pressed a "recall" button, which re-displayed the numbers which were programmed into the device. This was done to double-check the entry.[41]

After programming, the pharmacist simply pressed a "start" button and the compounder began operating. The pertinent issue in this case was whether the pharmacist should have observed the compounder while it automatically mixed the solutions to ensure that it was, in fact, operating properly. The court ruled that failure to observe the compounder was negligence to a degree necessary to warrant both punitive damages and aggravating circumstances damages.[42] Under Missouri law, punitive damages against a health care provider are appropriate for acts which are willful, wanton, malicious or so reckless as to be in utter disregard of the consequences.[43] Aggravating circumstances damages are appropriate when "defendant could have reasonably been charged with knowledge of a potentially dangerous situation but failed to act to prevent or reduce the danger."[44]

The Schroeder court considered a number of factors in determining the degree of culpability of the pharmacy. The manufacturer's user manual instructed the user to monitor the compounder at the start of the cycle and periodically thereafter during operation.[45] The pharmacist never read the manufacturer's manual, even though it was available to her.[46] The device contained alarms and indicators that were designed to alert the users to episodes requiring operator attention.[47] According to the pharmacist's testimony, the machine should have sounded an alarm if the pump wheel was not turning, but failed to do so.[48] Prior to this incident, the same pharmacist had prepared the same formulation, using the same compounder, 150 times with no untoward consequences.[49] This time, after pressing the "start" button, she turned her back to the device, but could hear it running.[50] She did not, however, observe the decrease in volume of the component solutions.[51] Because the prepared solution was colorless, she could not discern the lack of dextrose component by visual inspection.[52] Pharmacy policy did not require observation of the compounder during operation or end-testing of the prepared solution.[53]

After Schroeder's death, the pharmacy replaced the compounder with another, which malfunctioned twice.[54] These malfunctions were detected by the operator because she was observing the device during operation.[55] An expert witness for the defense testified that computerized biomedical equipment can malfunction frequently and the compounder was capable of malfunctioning even if it had been properly programmed.[56]

The Schroeder case provides a good example of the legal issues that should concern the user. The court considered the instructions and limitations for use provided by the manufacturer, the impossibility of programming a fail-safe device, and the duty the user has to consider the potential for severe harm if the computerized device malfunctions. Neither the manufacturer of the device, nor the computer programmer, were parties to the suit. In essence, the user was held liable for the consequences of the device malfunction.

### C. Injury

A software-driven medical device may injure a patient directly or indirectly. An example of direct injury is the application of excessive radiation during cancer therapy.[57] Examples of indirect injury are the failure to diagnose and treat, delay of treatment, or the administration of inappropriate treatment based upon erroneous computer data.[58] In a medical setting, the person most likely to be injured directly or indirectly by the software is the patient, who usually is neither the purchaser nor user of the medical device.

For the buyer of computer software in a medical setting, a wide range of economic damages are possible, if the purchased software does not live up to its expectations. The following are possible ill effects of an ineffective software package or computer system:[59]

1. Loss of value in resale.
2. Maintenance costs.
3. Site preparation and installation costs.
4. Acceptance testing, system training.
5. Hiring of additional personnel.
6. Purchasing of additional equipment.
7. Hiring of outside services to compensate.
8. Money spent on computer supplies, financing costs, value of lost personnel and management time, and business interruptions.
9. Consequences such as inability to meet regulatory requirements, contract commitments or other legal requirements resulting in fines, penalties, compliance costs, lawsuit costs and damage awards.

Before the avenues and methods of imposing liability for the injuries listed above can be discussed, we must first review the origins of the software errors that lead to the injury.

## V. DIFFICULTY IN MEDICAL PROGRAMMING

It is impossible to create an error-free program for medical use because of the unlimited combinations of clinical parameters.[60] No amount of testing can anticipate every possible situation that may arise when a large number of test subjects have their clinical data entered into the system.[61] Errors can be introduced in any of the developmental steps. Errors can also occur where the algorithms and coding are correct, but the design of the program fails to provide sufficient checks or duplication of critical operations.[62]

Errors may also be introduced into software system by the user. The programmer should minimize the possibility of user interface errors. The user may commit any combination of the following four types of errors:[63]

1. No response: a user might not respond if the user does not understand that the program requires a response.

2. Inappropriate response: user hits wrong key.

3. Incorrect recognition: if the user does not notice a hazard in the program.

4. Inappropriate timing: if the user responds too quickly or too slowly. (The program should have adequate timing margins.)

After considering the nature, origin and effect of software errors, we may now discuss legal efforts to deter and compensate victims of software malfunctions, and regulatory efforts to prevent the proliferation of faulty medical programming.

## VI. LEGAL EFFORTS TO DETER SOFTWARE MALFUNCTIONS

Common law offers two primary avenues of recovery for a party injured by a software malfunction or defective software design. If an express or implied contract exists between the programmer and the injured party, remedies under contract law may be available.[64] If there is no contract between the parties, traditional tort doctrines may apply.[65] A tort is a violation of a duty imposed by general law or otherwise, based upon of nature of the relationship of the parties in a given transaction.[66]

The first step in analyzing a potential claim under either contract law or tort doctrine is to determine whether the software provided by the vendor constitutes a "good" or a "service." If the software can be classified as a good, the contract provisions of the Uniform Commercial Code (UCC) will apply.[67] Also, the tort doctrine of strict product liability is theoretically possible.[68]

In determining whether a software package is a good or a service, courts have considered the following factors: whether the vendor included software in the sale of hardware,[69] the level of customization of the software,[70] the degree of service provided incidental to the purchase[71] and the proportion of financial payments attributed to goods or service.[72] In general, the inclusion of software within hardware favors a "goods" determination.[73] Customization of the software, and the purchase of extensive customer support favor a "service" determination.[74]

### A. Contract Law

If an express contract exists, the purchaser may sue for breach if the software fails to perform up to contractual specifications.[75] If a "goods" determination regarding the sale has been made, the purchaser may assert a breach of implied warranties of merchantability or fitness for use, under the UCC.[76] Suing under contract has an advantage over tort law because the plaintiff can recover the wide variety of pecuniary losses,[77] if they fall within the realm of the anticipated bargain.

In the medical setting, the patient rarely has a contract with, or is even aware of, the software developer. Usually, a computer-controlled medical device is purchased from the vendor by the health care facility, such as a hospital. For these reasons, contract law offers little chance of remedy against the programmer for an injured patient, and tort law becomes the preferred avenue for a plaintiff injured by a defective computerized medical device.

### B. Tort Law

If the software is classified as a "service," then traditional principles of negligence may apply. If the software is classified as a "good," an action in strict product liability is possible. There are inherent difficulties in the application of either of these doctrines to medical software malfunctions.

#### 1. Negligence

In order to sustain a cause of action under negligence, the plaintiff must prove four elements: duty, breach, causation and damages.[78] Each of these elements raise unique issues when applied to software malfunctions and defective software design.

##### Duty

It is not always evident who owes the duty to the patient. For example, it could be the product manufacturer, the systems analyst, the programmer, or all three.[79] Also, in many instances the plaintiff may be a third party unforeseen by the programmer, such as a patient.

Once the person who owes the duty is determined, much dispute remains as to the appropriate standard of care. Should a professional standard of negligence be applied to the computer programmer?[80] At least one court has so decided,[81] and several legal commentators

have discussed a new tort of computer malpractice.[82] However, one court has rejected the malpractice doctrine,[83] and absence of uniform standards of accreditation, lack of self regulation, and the lack of specified educational requirements make the emergence of a computer malpractice tort unlikely in the near future.[84]

### Breach

A software programmer might breach his duty by failing to write or test a program properly.[85] Because one can never write a perfect program or find all of software's flaws by testing, this type of breach depends upon the adequacy of the testing.[86]

A software programmer might breach his duty by failing to correct significant bugs.[87] However, if the user failed to inform the developer of the user's needs, failed to report any discovered defects in the program accurately and promptly, or failed to report any changes in the conditions of use that might warrant a software revision, and if the software is customized, the defendant may raise defenses of assumption of risk or comparative negligence.[88] The software vendor may breach his duty by failing to warn of limitations of use or by failing to instruct users how to operate the program.[89]

### Causation

Demonstrating that the injury resulted from an error in the software can be a difficult task, because injury may result from use of a computer even when the software is correct. Malfunction of a hardware component, an electrical surge, a user error, or any number of environmental conditions might cause the computer to fail.[90] The FDA acknowledges this problem by requiring computerized device manufacturers to evaluate environmental influences and random component failures and, if warranted, to design a redundancy between electro-mechanical and software controls.[91] This is done to reduce the potential for software malfunctions due to environmental factors or component failures.[92]

In the case of a computerized medical device, the injured plaintiff must first demonstrate that the harm was caused by a malfunction of the medical device, rather than by negligence of the medical staff, or any other possible cause. Once the plaintiff determines that the device did malfunction and cause the injury, the plaintiff must then assess whether or not a software component was a factor in the device's malfunction.

### Damages

If the injury due to a software error is indirect, such as a misdiagnosis due to erroneous lab results, actual damages caused by the erroneous computer value may be difficult to access. The physician's failure to run another test or to compensate with his own professional judgment may constitute an intervening cause.[93]

## 2. Strict Liability

Several legal commentators have offered strict liability as a means for redress against a manufacturer of a defective software product.[94] Application of this doctrine requires that the software package be classified as a "good" rather than a "service." One legal commentator has asserted that strict product liability should not be applied to physical injury cases involving faulty medical software.[95] Although imposition of the doctrine might encourage prolonged software testing and accountability, the risk of utilizing an occasionally unreliable software must be weighed against the cost of not applying software to medicine at all.[96] A comparison can be made to the court's reluctance to impose strict liability upon medical services.[97] Strict software liability would delay innovation and deny society of the many benefits incurred when software functions properly.

Because traditional negligence is difficult to prove, and strict liability is burdensome upon

society, it is better to minimize medical software malfunctions during the development, manufacture and sale of computerized devices, rather than by legal efforts to impose liability after the injury occurs. In pursuit of this goal, the Food and Drug Administration has issued a draft policy on the regulation of computer use in medicine[98] and a draft guideline for the review of computer software design in computerized medical devices.[99]

## VII. FDA REGULATION OF MEDICAL SOFTWARE

The FDA is authorized to regulate the use of software in the health care industry only to the extent that the software or the device containing the software falls under the definition of a medical device.[100] Assessing the applicability of FDA regulations to a computerized product is a 2-step process. First, one must determine if the product is of the type the FDA will regulate. If so, one must then determine the applicable level of regulation.

### A. FDA Policy

Certain medical uses of software are exempt from regulation. The FDA does not regulate computer products used for storage, retrieval and dissemination of medical information.[101] Nor does the FDA regulate computer products used for accounting, communications or educational purposes.[102]

Computer products which are medical devices are subject to one of four levels of control.[103] At the lowest level of control, the FDA imposes only the requirements for adulteration and misbranding.[104] This level of control applies to "general purpose articles" such as a personal computer programmed by a clinical chemist or a database management system with no medical claims.[105] These computers are not labeled or promoted for medical uses but are defined as medical devices by the nature of their use.[106]

The FDA's lowest level of control also applies to customized software used only in the physician's institution, such as an electronic bulletin board.[107] Computer products used in teaching and non-clinical research also fall into this category, as long as the research and development has not progressed to the level of human experimentation.[108] Although traditional library and educational activities are not regulated by the FDA, the use of computers for these functions still raise noteworthy legal issues. Before a medical facility upgrades to a computerized record system, they should consider both the evidentiary status of computerized records and the licensing requirements under their appropriate state law.[109] Because computerized medical records can rapidly be disseminated en masse, security breaches could be catastrophic.[110] Also, the long term storage capability of computerized disks must meet all applicable record retention requirements.[111]

The second tier of regulatory control applies to previously unclassified computer products such as "expert" or "knowledge based" systems, artificial intelligence and decision support systems.[112] Manufacturers of these devices may apply for exemptions to many of the FDA's regulatory requirements on a case by case basis.[113] This level of scrutiny does not apply to manufacturers of computer software intended for use in blood banks.[114]

The third level of control applies to those products requiring pre-market notification.[115] These software products are substantially equivalent to currently marketed devices or devices marketed before the 1976 Medical Device Amendments. This category includes most computerized versions of non-computerized devices. For these products, the manufacturer must register with the FDA, list their products, notify FDA prior to marketing, and meet any other requirements of their device class.[116]

The highest level of regulatory control is applied to products for which the FDA requires the manufacturer to demonstrate both the safety and efficacy of the device prior to market introduction. In general, these devices support or sustain human life,[117] are implants,[118] or present a potential and reasonable risk of illness or injury.[119] For these devices, the manufacturer must submit a Pre-Market Approval Application demonstrating the safety and efficacy of the device to the FDA.[120] This application must be formally approved by the FDA before the

device can be marketed. The application must be updated annually,[121] and modifications that might affect the safety or efficacy of the device cannot be made to the approved device without notification and review by the FDA.[122]

B. FDA Classifications of Computerized Medical Devices

The FDA has issued no software guidelines for the highest level of scrutiny, i.e., those devices requiring pre-market approval. The agency has, however, issued a reviewer's guide for the second highest tier of scrutiny, i.e., for computer controlled medical devices undergoing pre-market notification. In 1989, a government report written by the Committee on Science, Space and Technology investigations and oversight subcommittee described federal regulation of software as "more an art than science."[123] The report stated that, "[I]deally, an agency would like to apply a standard analysis to a software-driven system and receive an unambiguous evaluation of its potential for creating hazards. Such tools do not now exist."[124]

Perhaps in response to this statement, the FDA has focused its requirements for computerized devices on the evaluation and prevention of potential hazards. For devices undergoing pre-market review, the FDA requires the manufacturer to first establish a "level of concern" for the device. The level of concern is a function of the nature of the device, its intended use, and the severity of harm that the device could inflict, or permit, as a result of failures.[125]

Determination of the level of concern is a two step process. First the manufacturer must evaluate the overall device by considering its intended use, its degree of influence on therapy and diagnosis, and any potential hazards.[126] In this step of the analysis, the manufacture must disregard any control features or safety functions designed into the device.[127] As a result of this evaluation, a level of concern is established for the overall device.

In the second step, the manufacturer must focus on the device's software components. The firm must consider how the software is "directly or indirectly involved" in the factors that were considered in evaluation of the overall device.[128] The FDA asks "[A]re some components of the software independent of others and/or do they have little or no adverse affects while others play a more active role in controlling a delivery, diagnostic or monitoring function? Could some units of the software indirectly control or affect a delivery, diagnostic or monitoring function of the device even though the unit does not directly control those aspects of the device?"[129] After considering these factors, the manufacturer then assigns an appropriate level of concern to each software component. The highest level of concern for any software component can be no higher than the level of concern for the overall device.[130]

A primary component in the determination of a device's level of concern is the "hazard analysis." The hazard analysis can be performed at the overall device level or at the software level.[131] The hazard analysis at both levels contains the following elements: the hazardous event, the cause, the method of control, the corrective action, and the level of concern.[132] At the software level, examples of corrective action are fail-safe mechanisms, redundant controls, error-handling routines, fault-tolerance, alarm testing activities, etc..[133] If software is used to compensate, mitigate or warn of an overall device hazard, that specific hazard should be documented in the hazard analysis.[134]

The FDA defines three levels of concern: major, moderate and minor. The industry trade association, Health Industry Manufacturer's Associate (HIMA), proposed a fourth level of concern, "negligible," for software components not involved in controlling any potential hazard,[135] but the FDA declined to adopt this proposal.[136] The FDA's definitions of the three levels of concern are as follows:

> MAJOR: The level of concern is major if operation of the device or software function directly affects the patient so that failures or latent design flaws could result in death or serious injury to the patient or if it indirectly affects the patient (e.g., through the action of a care provider) such that incorrect or delayed information could result in death or serious injury of the patient. (emphasis added).

MODERATE: The level of concern is moderate if the operation of the device or software function directly affects the patient so that failures or latent design flaws could result in minor to moderate injury to the patient, or if indirectly affects the patient (e.g., through the action of a care provider) where incorrect or delayed information could result in injury of the patient. (emphasis added).

MINOR: The level of concern is minor if failures or latent design flaws would not be expected to result in death or injury to the patient. This level is assigned to a software component that the manufacturer can show to be totally independent of other software or hardware that may be involved in a potential hazard and should not directly or indirectly lead to a failure of the device that could cause a hazardous condition to occur. (emphasis added).[137]

When different levels of concern are assigned for different components of software, the highest level of concern assigned to any component becomes the level of concern assigned to the software of the device.[138] This software level of concern determines the degree and nature of the testing, development and documentation required for approval by the FDA.

### C. FDA's Perception of Software Development

The FDA acknowledges five phases of software development: specifications/requirements, design, implementation, verification and validation, and maintenance.[139]

Once the software specifications are established, the design phase entails "developing a conceptual view of the system, establishing system structure, identifying data streams and data stores, decomposing high level functions into subfunctions, establishing relationships and interconnections among components, developing concrete data representations, and specifying algorithmic details."[140] The FDA would like to see documentation regarding structure and processing details, modules and modularization criteria, design verification activities, and the testing objectives and test plan criteria at this phase.[141]

The implementation phase consists mainly of translating design specifications into source code. If done properly, debugging, testing and modification becomes easier.

The verification and validation phase includes testing performed at the unit, integration, and system level. The purpose of verification is to assure consistency between requirements, design, implementation and testing.[142]

The maintenance phase includes system enhancements or the correction of problems. These changes could occur at any level of the software development, and should be documented, verified, and traceable.[143] The manufacturer must have formal quality assurance procedures for implementing and documented any changes in the software of the device.[144]

### D. FDA's Documentation Requirements for Computerized Devices

The FDA requires documentation of the following software aspects: functional requirements and systems specifications, software design, software development, software verification, validation and testing, results, software certification and product labeling.[145] These aspects must be addressed for all software, regardless of its classified level of concern. However, the extent of the required testing and the details of the required documentation increase with a higher level of concern.[146]

Under functional requirements and systems specifications, the FDA looks for issues such as conflicts between overall device requirements and software requirements, bases in application or literature for algorithms, identification of software that controls the device's safety, identification of safe and unsafe operating states of the device, whether hazards are monitored directly or limited by software, whether the safety measures address environmental influences and random component failures, and whether measures exist for detecting errors at installation or during operation.[147]

Under the software development process the FDA examines procedures for developing

quality assurance and documentation of computer code, division of code into functional segments (modularization), whether standards are the same for code developed by outside contractors, descriptions of responsibility of personnel, designation of responsibility for revisions, regression testing, discussion of design reviews, whether software failure analysis was conducted "top down" or "bottom up," whether software is structured into sensible functional segments, whether there has been rigorous testing of built-in safety features, and whether results make sense in comparison to the similar device currently on the market.[148]

Under verification, validation and testing, the FDA queries whether tests are traceable back to system and software requirements, whether software is evaluated at module level prior to integration, whether tests demonstrate that integrated subunits interact properly, whether stress tests have been conducted, whether revisions are evaluated to assure they do not adversely affect other aspects of software. The FDA also examines the input and performance range for testing, and whether test strategies are sufficient to compare to the device currently on the market.[149]

Under test results, the FDA determines if the planned approach to designing, coding and testing is followed, if results demonstrate conformance at each level of software architecture (module/integration/system), and if tests try marginal data, incorrect commands and data, conditional sequences, momentary power failures and malfunctioning data sensors.[150]

Certification is merely a statement by an authorized individual in the organization that the software development process was followed, that quality assurance procedures were adhered to, that testing demonstrates that functional requirements were met, and that system specifications were fulfilled.[151] The Health Industry Manufacturer's Association (HIMA) believes that special signatures for software processes are "neither necessary or appropriate".[152] However, the requirement of certification will alleviate the difficulty in identifying the particular individual with the legal duty to properly develop and test the device's software.[153]

The FDA requires the following contents for labeling of computerized medical devices: adequate instructions for use, special requirements and considerations of use, instructions for response to fault or failure condition, likely causes of faults or failures, how the information is presented to user, and directions if user installation is required.[154]

From labeling, the user should be able to determine if software is appropriate for given task or intended use, if software will work with available equipment, and if purchaser can initiate use simply by consulting manual or instead must be trained. If training is necessary, the labeling should provide training requirements. If the user installs the software, instructions must be adequate. An installation qualification/checkout should be performed, regardless of who installs.[155]

Probable sources of software and hardware faults should be identified in labeling, including guidance for operator response. The information regarding potential faults should be readily accessible to user by means of visual or audible alarms, displays, or attached operator's manual or index guide. If hazards cannot be obviated by design, they must be identified and proscribed in warnings, either on the screen or in labeling.[156]

As demonstrated in the Schroeder case,[157] accurate labeling can shield the vendor from liability and transfer the responsibility for software failure upon the user. Accurate labeling can also thwart any action for strict product liability based upon inadequate warnings or instructions for use. However, FDA's labeling requirements take the concept of foreseeable circumstances to an extreme by requiring the manufacturer to document in writing all potential mishaps that might occur during use of the device. Unlike pharmaceutical labeling, where all potential adverse side effects are determined by clinical study, the potential hazards listed in device labeling may represent mere conjecture on the part of the manufacturer, based upon the firm's internal pre-market hazard analyses.

## VIII. PROPOSED PLANS FOR THE SUCCESSFUL DEVELOPMENT OF SOFTWARE

In response to the extensive regulation conducted by the FDA over the manufacturers of computerized medical devices,[158] software developers must enhance the sophistication of their software development techniques. Commentators have made suggestions how to minimize the possibility of legal problems arising from faulty software design.[159] These suggestions are highly

pertinent to the developer of medical software.

One team of legal commentators acknowledge that no generic set of standards can be applied to software development.[160] However, they offer suggestions that may apply under certain circumstances.[161]

1. Use qualified software programmers.
2. Document each step in the process, to show who wrote the program and the care taken in each step.
3. Test the software adequately and thoroughly before its release.
4. It may be prudent to have a different group of employees test the software than the group that developed it.
5. Take actions to avoid errors in the substantive information incorporated into the program. The vendor should keep records of how such technical information is incorporated, and who provided it.
6. If the software is being developed for a particular industry and there are applicable professional codes of conduct for that industry, it may be prudent to follow these codes.

These suggestions are in alignment with FDA's requirements and expectations. They recommend reasonable assignment of responsibility, a thorough documentation of the development process, adequate testing prior to release into the market, the existence of a separate quality assurance department to test the software, control over input parameters, and conformance with all applicable regulations.

Another commentator has recommended 6 steps that a firm should take to improve their software capabilities.[162]

1. Understand the current status of [the manufacturer's] development process or processes.
2. Develop a vision of the desired process.
3. Establish a list of required process improvement actions in order of priority.
4. Produce a plan to accomplish the required actions.
5. Commit the resources to execute the plan.
6. Start over at step 1.

Each of these steps, quoted above, are philosophically aligned with the requirements and expectations of the FDA. They concur with the FDA's conclusion that an understanding of development process or processes is critical. They promote the establishment of software specifications at an early and critical stage. They acknowledge that plans should be developed to implement required actions and to conform to specifications, and they realize that a conscientious effort in terms of resources and time must be made by the software developer to accomplish these goals.

## IX. CONCLUSION

At some time in our life, each of us will find ourselves a patient in the health care system. The potential for harm to an innocent patient due to software failures is alarming. Medical software has malfunctioned, and will continue to malfunction due to the nature and complexity of its use in the medical field. In general, software systems are being used in increasingly sensitive applications, and are becoming larger and more complex.[163] Unless we improve our error rates, proliferation of software usage will increase the chance of software error,[164] thereby increasing the chance of injury.

Under certain conditions, common law doctrines may provide redress to the injured party. However, a more hopeful solution is the routine development of less threatening and more reliable software by means of pre-planned specifications, thorough testing, hazard analyses, software documentation and review. In response to the severe potential for death or serious injury due to

even a single medical software failure, manufacturers and users of computerized medical devices must take a leadership role in society's efforts to enhance the reliability of computer software.

## ENDNOTES

1. Diane B. Lawrence, Strict Liability Computer Software and Medicine: Public Policy at the Crossroads, XXIII **Tort Ins. L.J.** 1, 1 (1987).

2. Arthur W. Hafner, Ph.D., Computers and the Legal Standard of Care, **Arch. Ophthalmol.** 1989; 107: 966.

3. Clement J. McDonald, M.D., William M. Tierney, M.D., Computer-Stored Medical Records: Their Future Role in Medical Practice, **JAMA(R)** 1988; 259: 3433-3440.

4. Deborah K. Fulton, Legal Problems Arising In the Automation of Medical Records, **Topics in Health Record Management** 1987; Volume 8, Number 2, at 73-79.

5. R. Brian Haynes, M.D., Ph.D., Cynthia J. Walker, MLS, Computer-Aided Quality Assurance: A Critical Appraisal, **Arch. Intern. Med.** 1987; 147: 1297-1301.

6. Edward H. Shortliffe, M.D., Ph.D., Computer Programs to Support Clinical Decision Making, **JAMA(R)** 1987; 258: 61-66.

7. Id.

8. Lawrence, supra note 1.

9. Id.

10. FDA Policy for the Regulation of Computer Products, Center for Devices and Radiological Health, Food and Drug Administration, (11/13/89) (draft) (revision of previous draft dated September 1987).

11. A recall is defined by Medical Device Regulations as a "firm's removal or correction of a marketed product that the Food and Drug Administration considers to be in violation of the laws it administers and against which the agency would initiate legal action, e.g., seizure. Recall does not include a market withdrawal or a stock recovery." 21 C.F.R. § 7.3(g) (1985). FDA may request or demand a recall, or the firm may initiate a recall on its own volition. 21 C.F.R. §§ 7.45-7.46. Even when a recall is voluntary and initiated by the firm, the firm is required to inform the FDA. 21 C.F.R. §7.46. The FDA, in turn, is required to promptly notify the public of all recalls, unless "public notification may cause unnecessary and harmful anxiety in patients and that initial consultation between patients and their physicians is essential." 21 C.F.R. § 7.50.

12. Westlaw, HTHT-FDA database, search for "computer or software," search commenced November 1, 1992; Lexis, ZMH1 database, search for "computer or software," search commenced May 30, 1995.

13. Class I recalls, where FDA has determined there is a "reasonable probability that the use of, or exposure to, a violative product will cause serious adverse consequences or death." 21 C.F.R. § 7.3(m)(1)(1985).

14. Class II recalls, where FDA has determined that "use of, or exposure to . . . [the device] may cause temporary or medically reversible adverse health consequences or where the probability of serious adverse health consequences is remote." 21 C.F.R. § 7.3(m)(2)(1985).

15. Class III recalls, where FDA has determined that "use of or exposure to . . . is not likely to cause adverse health consequences." 21 C.F.R. § 7.3(m)(3)(1985).

16. 90-22 FDA Enf. Rep. 5, 1991 WL 263118 (F.D.A.), May 30, 1990.

17. 92-1 FDA Enf. Rep. 11, 1992 WL 423 (F.D.A.), January 2, 1992.

18. 87-22 FDA Enf. Rep 16, 1987 WL 96704 (F.D.A.), June 3, 1987.

19. 88-26 FDA Enf. Rep. 7, 1988 WL 221699 (F.D.A.), June 29, 1988.

20. 90-3 FDA Enf. Rep. 11, 1990 WL 262718 (F.D.A.), January 31, 1990.

21. 91-44 FDA Enf. Rep. 22, 1991 WL 228975 (F.D.A.), October 30, 1991.

22. 91-52 FDA Enf. Rep 29, 1991 WL 279093 (F.D.A.), December 26, 1991.

23. 90-1 FDA Enf. Rep. 27, 1990 WL 262689 (F.D.A.), January 3, 1990.

24. 92-13 FDA Enf. Rep 25, 1992 WL 59884 (F.D.A.), March 25, 1992.

25. 86-30 FDA Enf. Rep. 6, 1986 WL 59665 (F.D.A.), July 23, 1986.

26. 91-15 FDA Enf. Rep. 29, 1991 WL 115754 (F.D.A.), April 10, 1991.

27. 92-13 FDA Enf. Rep. 31, 1992 WL 59890 (F.D.A.), March 25, 1992.

28. 91-17 FDA Enf. Rep. 5, 1991 WL 124637 (F.D.A.), April 24, 1992.

29. 90-50 FDA Enf. Rep. 26, 1989 WL 235817 (F.D.A.), December 13, 1989.

30. 86-47 FDA Enf. Rep. 3, 1986 WL 60058 (F.D.A.), November 19, 1986.

31. 89-27 FDA Enf. Rep. 3, 1989 WL 235356 (F.D.A.), July 5, 1989.

32. See infra, Part VII.B.

33. Julie Foreman, Computers in Clinical Medicine Raise Questions of Liability, **Arch. Ophthalmol.** 1989; 107: 25, citing Miller RA, Schaffner KF, Meiser A, Ethical and Legal Issues Related to the Use of Computer Programs in Clinical Medicine, **Ann. Intern. Med.** 1985; 102: 529-536.

34. In Helling v. Carey, 519 P.2d 981 (Wash. 1974), the Supreme Court of Washington held two ophthalmologists liable for medical malpractice because they failed to perform a glaucoma test on a patient who was under 40 years old. In neglecting to conduct the test,

the physicians followed the standard practice for the profession of ophthalmology. (The incidence of glaucoma in patients under 40 years old was 1 out of 25,000.) Because the glaucoma test was simple, inexpensive, informative and essentially harmless, the court imposed a duty to conduct the test in all instances, regardless of the current custom. In The T.J. Hooper, 60 F.2d 737 (2d Cir. 1932), the second circuit imposed a duty on tugboat owners to install radio receivers on their crafts even though most tugboats at that time were not equipped with receivers. The court said, "a whole calling may have unduly lagged in the adoption of new and available devices." Sixty years later, everyone would agree that the sailing of a tugboat without a radio receiver would be foolhardy. Similarly, the day is forthcoming when a physician shall be negligent if she fails to resort to a computerized check or consultation before implementing her medical decision.

35. See supra, Part III.

36. Schroeder v. Lester E. Cox Medical Center, Inc., 833 S.W. 411 (Mo. Ct. App. 1992).

37. 833 S.W.2d 411 (Mo. Ct. App. 1992).

38. Id. at 416.

39. See Id. at 418 (the erroneously prepared solution contained less than 10 milligrams per deciliter of dextrose; 5,000 milligrams per deciliter of dextrose was required).

40. Id. at 412.

41. Id. at 416.

42. Id. at 411.

43. Id. at 420.

44. 833 S.W.2d 411 (Mo. Ct. App. 1992), citing Dougherty v. Smith, 480 S.W.2d 519 (Mo. Ct. App. 1972).

45. Id. at 416.

46. Id.

47. Id.

48. Id. at 418.

49. 833 S.W.2d 411 (Mo. Ct. App. 1992) at 417.

50. Id.

51. Id.

52. Id.

53. Id. at 417.

54. Id. at 418.

55. 833 S.W.2d 411 (Mo. Ct. App. 1992) at 418.

56. Id. at 419.

57. See supra, note 18.

58. See supra, note 26.

59. Lanny J. Davis and J.T. Westermeier, Computer Contract Disputes- Supplemental Materials, C601 **ALI-ABA** 37, May 10, 1991).

60. Lawrence, supra, note 1.

61. Vincent N. Brannigan, J.D, and Ruth E. Dayhoff, M.D., Liability for Personal Injuries Caused by Defective Medical Computer Programs, 7 **Am.J. Law & Med.** 123, 126 (1981).

62. Id.

63. L. Nancy Birnbaum, Strict Product Liability and Computer Software, 8 Computer/L.J 135, 153 (1988)).

64. See, e.g., RRX Industries, Inc. v. Lab-Con, Inc., 772 F.2d 543 (9th Cir. 1985).

65. See, e.g., Diversified Graphics, Ltd. v. Groves, 868 F.2d 293 (8th Cir. 1989).

66. See Black's Law Dictionary 1489 (6th ed. 1990).

67. UCC § 2-102 (1989).

68. Lawrence, supra, note 1.

69. See Lawrence B. Levy and Suzanne Y. Bell, Software Product Liability: Understanding and Minimizing the Risks, 5 **High Tech L.J.** 1 (1990) (discussion of factors determinative in classification of software as a "good" or a "service").

70. Data Processing Services, Inc. v. L.H. Smith Oil Corporation, 492 N.E. 2d 314, 318-321 (Ind. Ct. App. 1986).

71. RRX Industries, Inc. v. Lab-Con, Inc., 772 F.2d 543, 546 (9th Cir. 1985).

72. Micro Managers, Inc. v. Gregory, 434 N.W.2d 97, 100 (Wis. Ct. App. 1988).

73. See Comment, Computer Malpractice and Other Legal Problems Posed by Computer "Vaporware", 33 Vill. L. Rev. 835, 855 (1988).

74. Levy and Bell, supra, note 69.

75. Davis & Westermeier, supra, note 59.

76. UCC §§ 2-314(1), 2-314(2)(3) and 2-315 (1989).

77. Supra, Part IV.C.

78. W. Page Keeton, et. al, Prosser & Keeton and the Law of Torts, §30 at 164-165 (5th ed. 1984).

79. Lawrence, supra, note 1.

80. See Diversified Graphic, Ltd. v. Groves, 868 F.2d 293, 295-297 (8th Cir. 1989) (discussion of "due professional care").

81. Data Processing Services, Inc. v. L.H. Smith Oil Corporation, 492 N.E. 2d 314 (Ind. Ct. App. 1986).

82. Levy and Bell, supra, note 69, Joseph P. Zammit, Tort Liability for Mishandling Data, Patents, Copyrights, Trademarks, and Literary Reports Course Handbook, 13th Annual Computer Law Institute, 322 PLI/Pat 429, PLI Order No. 64-3867, Joseph P. Zammit and Maria A. Savio, Tort Liability for High Risk Computer Software, Patents, Copyrights, Trademarks, and Literary Property Course Handbook Series, 9th Annual Computer Law Institute, 239 PLI/Pat 373, PLI Order No. G4-3802 (1987), and Sue Gake Graziano, Computer Malpractice- A New Tort on the Horizon?, 17 **Rutgers Computer & Tech. L.J.** 177 (1990).

83. Invacare Corp. v. Sperry Corp., 612 F.Supp. 448, 454 (N.D. Ohio 1984).

84. Joseph P. Zammit and Maria A. Savio, Tort Liability for High Risk Computer Software, Patents, Copyrights, Trademarks, and Literary Property Course Handbook Series, 9th Annual Computer Law Institute, 239 PLI/Pat 373, PLI Order No. G4-3802 (1987).

85. Levy and Bell, supra, note 69.

86. Id.

87. Id.

88. Zammit and Savio, supra, note 84.

89. Levy and Bell, supra, note 69.

90. Lawrence, supra, note 1.

91. FDA's Reviewer Guidance for Computer Controlled Medical Devices at 23.

92. Id.

93. An intervening cause is a cause of independent origin arising long after the programmer's negligence, and alleviating the programmer from legal responsibility for the consequences of the misdiagnosis. See, e.g., Keeton, et. al, supra, §144 at 309-319.

94. See Daniel T. Brooks, Professional Malpractice Liability of Computer Specialists, Patents, Copyrights, Trademarks, and Literary Property Course Handbook Series, 10th Annual Computer Law Institute, 259 PLI/Pat 707, PLI Order No. G4-3820 (1988), Joseph P. Zammit and Maria A. Savio, supra, note 84, Levy and Bell, supra, note 69, and Lawrence, supra, note 1.

95. Lawrence, supra, note 1.

96. Id.

97. cf Hoven V. Kelble, 256 N.W.2d 379, 391 (Wis. 1977).

98. FDA Policy for the Regulation of Computer Products, Food and Drug Administration, Center for Devices and Radiological Health, Food and Drug Administration, November 11, 1989 (draft) (hereinafter "FDA Policy").

99. Reviewer Guidance for Computer Controlled Medical Devices Undergoing 510(k) Review, Office of Device Evaluation, Centers for Devices and Radiological Health, Food and Drug Administration, August 29, 1991.

100. A medical device is defined as ". . . an instrument, apparatus, implement, machine, contrivance, implant, in vitro reagent, or other similar or related article, including any component, part, or accessory, which is . . . (2) intended for use in the diagnosis of disease or other conditions, or in the cure, mitigation, treatment, or prevention of disease, in man or other animals . . . (3) intended to affect the structure or any function of the body of man or other animals." 21 U.S.C. § 321 (h).

101. FDA Policy.

102. Id.

103. Id.

104. 21 U.S.C. §§ 351 and 352.

105. FDA Policy at 2.

106. Id.

107. FDA Policy at 2.

108. Id. at 3.

109. Deborah K. Fulton, Legal Problems Arising In the Automation of Medical Records, **Topics in Health Records Management**, Volume 8, Number 2, at 77 (1987).

110. Id.

111. Id. at 78.

112. FDA Policy at 3.

113. Id.

114. Id.

115. 21 U.S.C. §360(k).

116. FDA Policy at 3.

117. Life-supporting or life-sustaining is defined as a "device that is essential to, or that yields information that is essential to, the restoration or continuation of a bodily function important to the continuation of human life." 21 C.F.R. § 860.3(e) (1985).

118. Implant is defined as a "device that is placed into a surgically or naturally formed cavity of the human body and is intended to remain implanted continuously for a period of thirty days or more." The FDA may also determine that devices placed into the body for a shorter period of time are implants. 21 C.F.R. § 860.3(d)(1985).

119. FD&C Act § 513(a)(1)(C), 21 U.S.C. § 360c(a)(1)(c).

120. FD&C Act § 515, 21 U.S.C. § 360(e), 21 C.F.R. § 814.20.

121. 21 C.F.R. §814.84.

122. 21 C.F.R. § 814.39.

123. The "Gray Sheet," M-D-D-I Reports, 15(46): 11-12, November 13, 1989, citing "Bugs in the Program," Committee on Science, Space and Technology investigations and oversight subcommittee.

124. Id.

125. FDA's Reviewer Guidance for Computer Controlled Medical Devices at 3.

126. Id.

127. Id.

128. FDA's Reviewer Guidance for Computer-Controlled Medical Devices at 3.

129. Id.

130. Id. at 5.

131. Id. at 4.

132. Id.

133. Id.

134. Id.

135. Letter from Robert T. Schwartz, Director, Computer and Biomedical Technology Programs, Health Industry Manufacturer's Association to John C. Villforth, Director, Center for Devices and Radiological Health, Food and Drug Administration, November 1, 1989.

136. The FDA's August 29, 1992 guidelines contain no "negligible" level of concern. HIMA proposed that no documentation be required for the negligible level of concern. The FDA requires all software to be classified at least as "minor" in level of concern, and requires the manufacturer to provide a minimal level of documentation concerning the software.

137. FDA's Reviewer Guidance for Computer Controlled Medical Devices, at 4-5.

138. Id. at 5.

139. Id.

140. Id. at 7, citing Fairly, Software Engineering Concepts, McGraw-Hill, New York, 1985.

141. Id. at 7.

142. Id.

143. Id. at 7-8.

144. Id. at 8-9.

145. FDA's Reviewer Guidance for Computer-Controlled Medical Devices at 10-11.

146. Id. at 17-26.

147. Id. at 17-18, 20, 23-24.

148. Id. at 18, 20, 23-24.

149. Id. at 18-19, 21-23.

150. Id. at 19, 22, 24.

151. Id. at 19, 22, 24-25.

152. Letter from Robert T. Schwartz, Director, Computer and Biomedical Technology Programs, Health Industry Manufacturer's Association to John C. Villforth, Director, Center for Devices and Radiological Health, Food and Drug Administration, at 3 (November 1, 1989).

153. Supra, Part VI.B.1.

154. FDA's Reviewer Guidance for Computer Controlled Medical Devices at 27-28.

155. Id.

156. Id.

157. Supra, Part IV.B.

158. The Health Industry Manufacturer's Association (HIMA) considers some of the detailed procedures specified by the FDA as "unnecessarily burdensome to the industry and FDA and to be potentially ineffective in determining safety, effectiveness, or substantial equivalence." HIMA in particular questions whether FDA should evaluate a device's development process. See letter from Robert T. Schwartz, Director, Computer and Biomedical Technology Programs, Health Industry Manufacturer's Association to John C. Villforth, Director, Center for Devices and Radiological Health, Food and Drug Administration, November 1, 1989, p. 1.

159. See, e.g., Suzanne Y. Bell and Lawrence B. Levy, Software Product Liability: Understanding and Minimizing the Risks, 4 High Tech L.J. 1, 1990, and Watts S. Humphry, Managing the Software Process, Addison Wesley Publishing Co.

160. Levy and Bell, supra, note 69.

161. Id.

162. Watts Humphry, Managing the Software Process, Addison Wesley Publishing Company, p. 4.

163. Watts Humphry, Managing the Software Process, Addison Wesley Publishing Company, at 13.

164. Id.

# Developing Quality Processes in a Startup Team at ParcPlace
## *or*
## Process is Your Friend


## Abstract

Small engineering teams can benefit from establishing software quality processes and developing support tools. Software process works best if the team works together to develop and refine the process to create the one that works best for them rather than following guidelines set out in a book. This paper describes how a small team came together and invented a process to support their work. The team found that software development became smoother and less stressful, that they enjoyed a richer relationship with the customer, and that it was easier to produce a quality product. The paper focuses on the development of the process in addition to describing the process steps.



## Jay VanSant

Jay VanSant is Quality Assurance Manager at ParcPlace Systems. He has eleven years experience in the quality assurance field and is a Certified Quality Engineer. In addition to his expertise in software testing, Jay specializes in process improvement and team facilitation. At Mentor Graphics during the 1980s, Jay served as Corporate Release Manager and Continuous Improvement Process Manager. Jay holds degrees in psychology and mathematics and has done postgraduate work in computer science.

## Jenny Greenleaf

Jenny Greenleaf is a user interface designer at ParcPlace Systems. A thirteen-year veteran of the high-tech industry, her background is in communications and technical writing, including a stint in Apple's Advanced Technology Group. She is most interested in usability, information filtering, and simplifying complex interfaces. Jenny holds a degree in journalism and has done further coursework in human-computer interaction.


## The Team

Liesl Andrico
John Copp
Ward Cunningham
Mike Dolbear
Jack Falk
Belinda Flynn
Cynthia Gens
Philip Goward
Jenny Greenleaf
Bill Greenseth
Robert Greenspan
Elizabeth Hollomon
Verna Knapp
Ray Lischner
Paul McCullough
Bob Million
Mark Pirogovsky
Bruce Schneider
Lise Storc
Jay VanSant

## Introduction

New software engineering teams face special challenges in creating products. With no established history or processes, teams must invent the way they work as well as the product. This paper describes the experiences of a team who had to create their engineering environment from "scratch" and how the team evolved that process. The process and supporting tools enabled the team to focus on their customer and to create an enviable record of on-time, on-budget deliveries. Most importantly, the team came to value and enjoy the process it crafted.

Small software engineering teams often have difficulty seeing how information about process improvement and software engineering technique applies to them. We hope that this paper will help small teams understand that they can develop and support processes that emphasize customer focus, facilitate team communication and integration, and produce excellent software. The paper describes several phases that the team went through and provides commentary about the process of making process.

## Phase 1: Forming the team and understanding the customer's needs

ParcPlace Systems, which is based in California, wanted to staff a Portland, Oregon, office to work on a specific, long-range contract software project for a third party. In the fall of 1992, ParcPlace hired a very complete team consisting of a manager, a system administrator/office manager, seven engineers, and a technical writer who also took on user interface tasks (co-author Jenny Greenleaf). A QA manager (co-author Jay VanSant) was hired within about five months, and a course developer/trainer was hired somewhat later. Most of the team, which had an average experience level of over 10 years, started work within a month of each other. The team was given a very general goal: Figure out what the customer wants and build it. Very little additional direction was offered.

The client wanted software to support a proprietary business re-engineering methodology. The methodology was relatively new to them and not well understood or documented. Initially, the team spent long hours in meetings with the client's project manager and methodology visionary.

The methodology had evolved as a way of working with clients to model the work of their business in outlines, and they used an outline processor to capture and print these outlines. The client envisioned a tool that could use the outlines to generate prototype information system (database) applications. While the concept seemed straightforward, the team found that working with a visionary to develop requirements was a real challenge. It became clear that the first objective was to develop a prototype that would embody the initial ideas.

The functional elements of the prototype were defined, and the engineers were invited to take whatever piece of it they wanted to work on and "just do it." The first prototype, code named *Marys Peak,* was built in about three months. The specification was written in parallel with the implementation, the process can best be described as ad-hoc, and the major method of coordination and control was impromptu meetings in hallways and offices. The understanding of the client's methodology continued to change right up to the very last, requiring repeated revisions of the software and specification. Integration was done manually, using just a directory structure on the UNIX server as the code configuration management system. There was no organized testing, although a defect-tracking tool was developed during this period.

The free-form approach was quite exhilarating at first, but quickly broke down, and friction developed between several of the team members. It was intense and frustrating, but a prototype was created and demonstrated to the client. The prototype provided a point of comparison, and the client firm was better able to articulate its vision. Both the engineering team and the client learned

more about the methodology while building the prototype, and several changes were made to the methodology as a result.

During development of the *Marys Peak* prototype, one of the engineers (Ray Lischner) designed and built a defect and change request tracking tool. He used the client's methodology to model the necessary tasks, which also gave him more insight into the client's methodology. He then designed and implemented the Defect Tracker to support these activities. Both management and the client understood that such a tool was essential to the success of the project and agreed to this use of a key engineering resource.

The team's QA manager (co-author Jay VanSant) joined the group at the end of the *Marys Peak* prototyping phase. The team had been advertising for a QA Engineer/Manager for several months, looking for a person who had experience in both process development and software testing. Jay's first tasks were to participate in a post-mortem review of Mary's Peak and to develop and propose a process model for the team.

## Reflections on Phase 1

- The experience of total "ad-hocracy" (with the resulting frictions and frustrations) convinced everyone on the team that some level of formal process was necessary and good. It may be that a similar experience is the only way that many software engineers can come to appreciate process, since the steps of good process are seldom really learned in Computer Science departments.
- Ray's initiative in developing the Defect Tracker led the way toward acceptance that it is necessary to devote some part of your engineering resources toward providing tools to support your own software engineering process.

## Phase 2: Understanding ourselves and the team process

Because what we had learned from the first prototype resulted in so many changes to the envisioned product, we decided to build a second prototype, code-named *Powell Butte*, to test out the new ideas before proceeding with the development of the real product. Before starting on the second prototype, however, we stepped back to examine what had happened while we were creating the first.

We knew that our process (or lack thereof) had been a problem, so we conducted a post-mortem to evaluate what had happened and to determine where we wanted to go and how we wanted to get there. The group first listed what had gone right. Once everyone was in a good mood regarding accomplishments, we got around to expressing dissatisfaction with almost everything! The team decided what could be done to address our problems and people were assigned to investigate and report on process issues ranging from scheduling to internal communications. Committees were formed to make recommendations on key issues, such as the role of a system architect, what sort of process we should use, and how to control changes. Even a "fun committee" was set up to investigate office environment and team building issues. Key to this effort was that everyone participated; we each had at least one, if not three, committee assignments.

One of the most significant efforts in this investigatory phase was Jay's interviews of each team member to find out about their past experiences and expectations about quality assurance (QA) and process. He asked the following questions:

- What are your past experiences with QA and process improvement?
- What is your attitude toward your responsibility for quality and testing?
- What are your specific attitudes toward process steps for problem prevention or early problem detection, such as reviews, walkthroughs, inspections, unit testing, integration testing, and others.
- What are your expectations of QA? (in this case, Jay!)
- What are your private views of how interaction with team members and management had worked during the initial phase of the project?
- What experience do you have with and what is your attitude toward processes and tools for project management, defect tracking, configuration management, and testing.
- What experience do you have and what is your attitude toward development of intermediate work products such as requirements documentation, project plans, specifications, design documentation, test plans, automated test beds, and others.

During these interviews, Jay shared his experiences and attitudes about each of the areas listed above, but mainly focused on the team member's views in a very non-judgmental way. The reason for the interviews was to gather information that would allow us to develop a process that we could all agree to implement.

Once the information had been gathered, and Jay had a good idea of people's attitudes and expectations, he set up meetings to develop a process proposal for the team. All were encouraged, but not required, to attend. In general, the people with strong views about process attended the meetings, and we discussed the inclusion of various process elements. Based on these discussions, Jay wrote a proposal outlining a product life cycle model that described the main phases of a project, the process and process metrics to be used in each phase, the work product outputs of those phases, and the ways to measure the "goodness" of those work products.

The full team reviewed the proposal and reached consensus on what we would accept and what we could implement in the next phase of product development. Most of the proposal was accepted without much change, and the final proposal was documented and distributed to all members of the team. Templates and review criteria were subsequently developed for intermediate work product deliverables. Team members agreed on a process that included steps for research, planning, design, implementation, testing/debugging, and post-release evaluation. (An outline of the process is provided in the appendix to this paper.) We also initiated communication improvements, such as establishing a project notebook and holding regular staff meetings. (Some did not consider the latter an improvement!)

In spite of all the furor surrounding process development, we also managed to develop and test the second prototype, *Powell Butte*. This prototype was delivered with documentation and training so the client's staff could learn to use it and simultaneously provide us with feedback on usability and potential enhancements. Feedback from the class was summarized and turned into the first version of a Functional Requirements Document. We all agreed to throw away the prototype code and begin development solely based on the new requirements.

## Reflections on Phase 2

- The only process that counts in any organization is the one that is actually USED by the team members. It is essential to understand the experience and attitudes of the individual team members, and to facilitate consensus around a process that incorporates the best of their current techniques along with improvements that offer clear benefits.
- The person filling each role in the team needs specific elements to do their job. It is important for all of the team members to understand what the other team members need. Development engineers and UI designers need to have clear, reasonably stable requirements to work from, and management that listens to them concerning schedules. QA engineers and writers need to have reasonably detailed specifications of the product as early as possible in the schedule and management that listens to them concerning schedules.
- Everyone appreciates a process with minimal overhead of meetings, status reporting, and documentation. The best process is the least process that still meets the needs of the customer and the team members. Process should be as simple as possible, and no simpler.
- Specifications are a pivotal step in good software engineering process. Having reasonably detailed specifications early in a software project provides the following benefits:
    - The discipline of clearly thinking through what needs to be built to meet the market's or customer's requirements helps engineers develop better architectures, interfaces, and realistic detailed schedules.
    - When a product is split up into different functional areas among a number of engineers, someone needs to take clear responsibility for user interface design and make sure that all parts of the product are consistent within the product and with platform standards.
    - QA engineers can create usage scenarios by looking at how each customer requirement can be accomplished using the specified user interface or API features. These usage scenarios can then be used to develop detailed test cases in parallel with detailed design and implementation, so that well designed tests are ready when the code is ready.
    - The specifications and usage scenarios allow writers to develop user manuals in parallel with design, implementation, and test. Drafts can then be ready early enough to allow thorough technical review before the final schedule crunch at the end of the project.
- Process metrics range from the simple to the elaborate. The simplest metrics consist of counting new, open, and closed defect reports each week and tracking planned versus actual completion of project tasks. This information is used each week to track progress toward completing a quality product.
- More elaborate metrics that track time spent against work completed (lines of code, function points, bugs fixed, etc.) may be useful in improving an organization's ability to estimate projects in the future. However, these more elaborate measures introduce a lot of overhead into the process, and often trigger fear and loathing among team members. In our project, such metrics were proposed and implemented, but were dropped later. It became clear that team members didn't want to keep the records carefully and were not comfortable with the way they had been defined and management really wasn't using the information anyway. We came to the conclusion that these types of metrics should be used only in situations that absolutely require it (e.g., the customer insists), and then only with clear guidelines and simple methods for collecting and using the information, and consensus among team members on following the process.

## Phase 3: Supporting and using the team process

The *Powell Butte* prototype was delivered along with a week-long training session given to the client's representatives. During this session, team members spent time with the new users to understand their reactions to the prototype and begin clarifying requirements for the product. We observed how the customers initially interacted with the user interface to learn where the interface was reasonably intuitive to use and where there were problems. We also interviewed the modelers and engineers who would use the product. These interviews, coupled with their experience using an actual prototype, enabled us to discover more of their requirements.

### *Closing the loop on requirements*

After completing the *Powell Butte* prototype and agreeing upon our process, work began on developing the first version of the actual product, code-named *Saddle Mountain.* We created an initial outline of requirements based on what we had learned in the *Powell Butte* training class, and review meetings were held to allow the client to comment on the list. Engineers thought about what corollary features were implied by the requirements, and so expanded and elaborated on the initial list. More review meetings were held.

Team meetings were held to look at how the work could be divided among engineers, and initial estimates were made about the time it would take to implement various features. The requirements list was prioritized, and a "cut-line" was drawn indicating what we thought could be done in the time frame that the client specified for the first release of the product. More review meetings were held to discuss the prioritization and cut-line. Finally, after 10 versions of the requirements outline were created and reviewed, we had what we felt were the "real" requirements. We held a formal inspection and incorporated the changes into the final eleventh version of the requirements.

### *Providing tools and training to support the process*

During this same period, work began on tools to support the process we had crafted. The process called for five key support tools: the development environment, a defect-tracking tool, a test development and regression testing framework, a configuration management environment, and a project management tool.

The most basic process support tool is the development environment itself. We used ParcPlace Systems' primary product, the VisualWorks application development environment, which provides the compiler, class libraries, and browsers for Smalltalk, a fully object-oriented language. It also provides graphical-user-interface-building tools that are relatively easy to use—enough so that our non-programming user interface person (Jenny) could create most of the windows, menus, dialog boxes, and error messages. The Smalltalk environment supports rapid development of object-oriented applications with graphical user interfaces and relational database connections, which is just what the we needed to build the next three process support tools.

The Defect Tracker initially was built using a file-based database. With VisualWorks 2.0 (which was just being released as a Beta version at this time), Ray enhanced it to work with an Oracle database. Everyone on the project used this system—engineers, writer, UI designer, test engineers, and managers.

With the help of interested engineers on the team, Jay developed a testing framework called TestWorks. It enables engineers to create test suites of test methods, run them interactively while logging results, and finally save baseline results with the tests so that they can be incorporated into regression test suites. Ray created a Test Coverage Analysis (TCA) tool that works with TestWorks.

Bill Greenseth created A Configuration Management Extension (ACME) tool that provided version control, configuration management, and a build procedure for the product. It allows developers to browse the differences between classes and methods in released versions of the product, and store "deltas" containing just the changes into the Oracle database. This tool is elegantly simple and easy to use; it doesn't have all the bells and whistles of a commercial CM system, but it also has none of the headaches.

Paul McCullough chose Microsoft Project to be the project management tool. He attended training for using it, and then used it to model the tasks and schedules for the *Saddle Mountain* project.

Robert Greenspan and Jay attended Inspection Facilitation training so there would be three engineers on the team who could conduct inspections on specifications, designs, and code. Ray already had attended the training and had led some early inspections.

### *Closing the loop on specifications and usage scenarios*

Philip Goward was selected to be the system architect for the Saddle Mountain phase of the BusinessWorks project. Based on a high-level architectural description of the product, the team divided up the work into functional areas assigned to each engineer. Each engineer worked on detailed functional specifications of their areas of the product, while Jenny created a user interface specification for the product and helped the engineers with the user interface portions of their specifications.

Jenny's UI spec provided a complete view of all the controls and menu items on each of the windows and dialogs in the product, along with descriptions of what actions each control or menu item had. Since product's functionality was organized around three types of outlines, the UI specification also provided detailed descriptions of how users interacted with the product to create outline headings, change the level of headings, move headings around, add attachments to headings, change views, print, save, and complete other tasks.

As soon as initial drafts of the specifications were published, Jay began working on usage scenarios describing how each of the listed requirements could be accomplished using the UI described in the specifications. The process of creating these usage scenarios uncovered many consistency issues and areas of incompleteness in the early specifications. When Jay decided he could describe how each of the requirements could be accomplished using the UI presented in the specifications, we knew that the specifications were complete relative to the requirements.

When the specifications and usage scenarios were complete, we sent copies of them to the client's managers for review. We received little feedback concerning them, probably because there was so much material to cover. Consequently, we scheduled a formal inspection to cover the UI specification and the usage scenarios and to make sure that the client's managers, modelers, and engineers would at least review these "overview" documents in detail. We took two days to cover both of these documents, inspecting each a page at a time. All comments were noted, and discussion was kept short and focused. Several key issues concerning the behavior of the product were found and corrected. At the end of the two days, we had confidence that we and our customer were in agreement about what we were building. Some new ideas had emerged for new features during the discussions, so there were some follow-on negotiations about priorities and schedules.

### *Implementing and testing Saddle Mountain*

With clear requirements and detailed specifications, the work of designing, coding, and testing the product proceeded quickly. Jenny had already created many of the actual windows, dialogs, and menus to use as screen shots for the specs. Since all of the functionality of the product focused around the behavior of the outliner, Verna Knapp wrote a design document describing the public interface for the classes and methods of the outliner. The other engineers on the team used these methods to integrate the various attachments and dialogs that added the functionality associated with automatically creating database application prototypes, displaying a tree view of the outlines, and other features.

We hired two contract QA engineers to help test the product. Belinda Flynn initially tried to interface VisualWorks with a GUI-testing framework called QA Partner, but the scripting extensions needed for dealing with the custom attachments in the outline proved impossible to implement. Therefore, most of the testing was done interactively, with some automated tests created in TestWorks. We used the usage scenarios document as a basis for the detailed test plans, annotating the scenarios with plans for boundary testing and error testing. Several of the engineers built regression tests using TestWorks and continuously upgraded these tests as they added new functionality.

Weekly builds of the product were done in ACME, after each of the engineers released their areas of the code. Each engineer owned certain categories of classes. If other engineers needed to add or modify methods in classes owned by another engineer, they would release their changes to ACME and inform the owner of the changes by e-mail. The owner of the classes would then integrate the changes before releasing the classes to ACME with the correct version label for the weekly build. When the weekly build was completed, an acceptance test consisting of a subset of the regression tests was run before declaring the build ready to be used as a new baseline.

Since each build contained code under development, testing following the test plan would often fail because of partially implemented code. Development engineers did not want to see defect reports on this code, so they would provide information with the builds (via e-mail) describing which areas of the functionality were ready for testing and which areas were not.

We also hired a contract technical writer to prepare the user's manual, freeing Jenny to concentrate on evangelizing usability and on keeping the interface consistent and attractive.

Each week, we would have two meetings. On Mondays, at the regular staff meeting, we would go around the room and each person would describe what they had done during the last week and what they were going to be working on during the next week. Team members had the opportunity to raise any issues concerning either the product or the process. In a separate meeting, usually on Monday afternoons, Jay would distribute lists of open defect reports. During this meeting, we would make sure that these reports were correctly prioritized and assigned to product areas for which specific engineers were responsible.

### *Closing the loop on the product, documentation, and training*

Our tech writer completed an early draft and the User Manual was reviewed internally and by our customers. Early (beta) releases of the product were sent to the client after most of the functionality was complete, so that they could evaluate the way it worked. However, we got little feedback from these early releases, since the client's modelers and engineers were not yet trained and did not give the tools much use. Lise Storc, hired at the beginning of 1994, developed training for modelers and engineers, and training sessions were scheduled following the release of the product. These training sessions provided our best early feedback from customers on their acceptance of the product.

## Reflections on Phase 3

- Software product quality is defined by the customers who use the product. There is no substitute for a close relationship with customer representatives during the early phases of product definition and design. In our project, we were fortunate in having a single customer organization willing to work closely with us during this stage of the process. We initially worked with the project manager and the methodologist, but as the ideas for the product began to emerge, we brought in the future users of the product: the modelers and engineers. It is very important to discover the different types of users of a product and identify their unique requirements.

  We were critical of the time it took to refine the requirements, since we fell into the trap of "creeping featureism" and then had to negotiate what would be dropped so we could meet our deadline. We learned that there is a time to say "good enough."

- We found it beneficial to have a designated user interface designer for the product. Staffing this role ensured a more consistent, usable, and platform-compliant interface. Since most of what the customer sees is the user interface, attention to this facet of design is essential. We also found that it was helpful to have a non-programmer as a user interface designer, since she didn't let implementation considerations influence her design choices.

- When customers (internal or external) are provided with documents for review, it is important to set up a direct meeting to gather the feedback. In some cases, an informal review or discussion of the main areas of the document will suffice. However, in the case of key documents, such as requirements, UI specs, and usage scenarios, a formal inspection process works well to provide the greatest amount of useful review information in the least amount of time. Training in facilitating inspections is widely available.

- The five key support tools described above were critical to minimizing chaos in the software development process. Sophisticated versions of these tools are available from commercial vendors. The choice of development environment is obviously a critical bet-your-company decision, and few engineering organizations scrimp in this area. Commercial project management tools are quite mature and reasonably inexpensive now, and are available for every platform and operating system. The obstacle, in most cases, is learning how to use them well.

  The other three tools (testing framework, problem tracking system, and configuration management tool) can be either purchased or built internally. In our case, it was reasonably straight forward to build them using the VisualWorks/Smalltalk environment, and the resulting tools were well tailored to the size of our group and the process we adopted. Either way you go, it is important to devote sufficient engineering and management time to integrating these tools with the process you're using and to train team members to use them productively.

- Usage scenarios (or use cases, as they are sometimes called) are most commonly described as a first step in analysis to discover the actors and services (and later, objects and methods) that need to be part of an object-oriented software design. However, we used them as a way of clarifying UI issues and determining the completeness of the UI in relation to established requirements. Here are the steps we use in our process:

  1. Create a list of requirements stated in terms of what users of the software will be able to accomplish.
  2. Create a draft UI specification outlining screens, menu items, and the detailed behavior of mouse and keyboard events on controls in the UI.
  3. For each requirement, ask yourself "How will the user accomplish this with the specified UI?".

4. Document the "how" by alternately describing an action taken by the user with elements of the UI, followed by responsive actions of the UI.

   The usage scenarios were written like a play with two actors, the user and the product. In some cases, you may need to have more than one type of user, and break the UI up into more than one "agent" responding to those users.

5. When you come to requirements that cannot be accomplished with the specified UI, or where the usage seems confusing, then change or extend the UI until all of the requirements can be met in a reasonably simple way by the proposed UI.

- We discovered that this process of writing usage scenarios has the following benefits:
  - It tests the completeness and usability of the proposed UI in relation to the requirements.
  - Customers can review them to see if the proposed UI will have the "look and feel" they want.
  - QA engineers can use them as a basis for detailed test procedures for either interactive or automated testing.
  - Technical writers can use them with the UI specification as a basis for writing the user documentation for the product.
  - Training developers can use them to design training slides and labs.
- During the implementation and testing phase of the project, it is important to have a regular build cycle and to provide a way for engineering to provide information about what is ready for test (and not ready for test) in each build. This prevents a lot of wasted time for both the QA engineers (testing and reporting bugs) and the development engineers (responding to bug reports). Regular, short, focused meetings are also important, to discuss what is done, what is next, and any problems that are being discovered.
- It is important to establish ways for getting customer feedback after the product is sent to them (either as beta, or after the actual product ship date). This is often hard to do, since customers often want to see the product as early as possible, but are not willing to really use it for production work until after it is fully stable and they are trained on it.

## Phase 4: Reviewing and improving the team process

After the *Saddle Mountain* release, we held a post-mortem review meeting. We first went over what went well, and then listed things which did not work so well, or aspects of the process that could be improved. This meeting mostly uncovered areas where execution of the steps of the process were weak, rather than problems with the documented process. The ideas for improvement can be grouped into the following areas:

- Improve project tracking, with greater visibility of status to the team.
- Improve test planning, do more testing by engineering, and improve TestWorks to automate GUI testing.
- Improve communication between engineers and their management.
- Improve the effectiveness of the system architect.

Task forces of team members interested in working on each of these issues met and planned actions.

- Liesl Andrico (system admin/office manager) took over the job of creating and updating the project plan, now using MacProject instead of Microsoft Project. She talked with each person at the end of each week and updated very visible charts on the wall.
- Detailed test plans were created in relation to each of the new feature sets specified for the next release (code-named *Sam Hill*). Work got underway on GUITester, a test class library that would provide automated GUI testing functionality in TestWorks.
- Further meetings were held to openly discuss the perceived problems between engineers and managers.
- Frank Adrian led a task force to better define the role of the system architect.

During the *Sam Hill* phase of the product, we implemented some features that had been deferred in the *Saddle Mountain* project, improved the usability of the product as a team tool, and implemented a few new features. We followed the process, creating and reviewing detailed requirements and specifications with our customer. However, these steps were done much more efficiently than the first time because both the team members and our customers understood the process we used.

## Reflections on Phase 4

- There is always room for improvement in the way we do things. Sometimes this means doing the same process with better execution of the steps. However, as organizations grow and the tasks and technologies used evolve, the process itself must evolve. Our process has worked well for a small startup team working with a single, cooperative client. As we get into working on more products designed for wider markets, we must invent new ways to identify the requirements of our customers and to hear feedback from them concerning what we have built. Every team should reflect on the value and effectiveness of the process steps they use at the beginning and end of each major project, and take steps to make what they are doing more appropriate for the needs of their customers and the mission of their business.

## Conclusions

This paper has described how a small team worked together to develop a process that facilitated software development and produced a quality product for our customers. The process of developing process was not painless! There were frequent disagreements and occasional hurt feelings. However, the team ended up with a process that works like a well-oiled machine. And, like a Japanese factory, everyone has the right to stop the line and question the process. The most important things we learned were:

- Everyone on the team needs to take part in defining the process. Total team participation leads to buy-in. If the team doesn't support the process, it won't use it.
- The process must be flexible. The team needs to be able to evolve and adapt the process as needed. The process must be a tool for the team; the team should not be a tool for the process.
- The process must meet each team member's needs for information, commitments, and support.
- Management must support tool development or purchase. You need good tools to build good products.
- Customer involvement is essential. Ultimately, everything has to be focused on meeting the customer's expectations.

# Appendix: Portland ParcPlace Systems Process

## Product/Project Phases and Milestones

### Phase: Analysis

1. Identify preliminary requirements
2. Define product parts and priorities
3. Communicate overall architecture
4. Prototype (experiment) to explore issues
5. Write detailed customer requirements
6. Write preliminary Project Plan

### Phase: Plan

1. Write Functional Specifications
2. Write Quality Plan
3. Write Project Plan
4. Document detailed product architecture

### Phase: Design

1. Write detailed design documents
2. Write test plans

### Phase: Implement

1. Write user documentation
2. Write code
3. Write and run unit test cases.
4. Write system test cases.

### Phase: Test & Debug

1. Integrate product
2. System test the product and document results
3. Manufacture pilot and install test it.

### Phase: After Release

1. Conduct post-mortem meetings and document lessons learned.
2. Make necessary changes based on lessons learned.
3. Monitor customer feedback on product quality.

# Software Testing in a Product Enhancement Team

Jonathan R. Brandt
Spectra-Physics Scanning Systems, Inc.
959 Terry Street
Eugene, Oregon 97402-9120
jonb@sp-eug.com

## Abstract

This paper describes an improvement effort and the software testing process that resulted. The process was developed by a team that develops software product enhancements and works to fix product problems found by customers. The products under test are barcode scanners which execute real-time embedded software. The team uses PC based test systems for automated testing that execute test scripts on top of an internally developed application. The paper concentrates on the process we developed and why the process items were important to the team. The paper describes project schedules, a development procedure, a process flowchart, a process checklist, a test system configuration, test plans, test scripts, test reports, weekly review meetings, and problem tracking. The paper also discusses the improvement process that we used.

## Keywords

Software Testing, Process Improvement, Product Enhancements, Test Plans, Test Scripts, Test Reports, Embedded Systems, ISO9001.

## Biographical Sketch

Jonathan Brandt is a Software Engineer at Spectra-Physics Scanning Systems in Eugene, Oregon. He earned B.S. degrees in Electrical Engineering and Biomedical Engineering from the University of Iowa in 1988. Previously, he was a Software Engineer at Motorola, Land Mobile Products Sector, in Schaumburg, IL.

## Credits

The team members who developed and implemented this process and provided helpful suggestions for this paper are: Peter Boon, Glen Davis, Spencer Doidge, Peggy Fitzgerald, Jeff Knowlton, Randy Pope, Nick Tabet, Larry Treinen, and Jonathan Brandt.

# 1. Introduction

More than a year ago, Spectra-Physics Scanning Systems, Inc. divided its engineering research and development organization into a Customer Engineering Support group and a New Products group. The New Products group develops a product until it is available for general shipment. Then they transfer the product to the Customer Engineering Support (CES) group for ongoing support and enhancement. This paper describes the software test process that was developed by the CES group and the steps that were taken to make improvements.

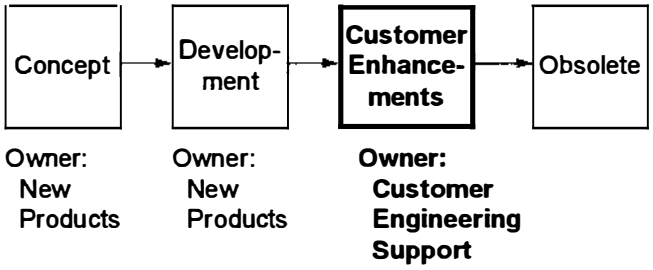
## 1.1 Product/Test System Description

Spectra-Physics Scanning Systems, Inc. develops barcode scanners. The company sells scanners for grocery, retail, and industrial applications. The scanners provide a product identification number to the customer's system through a variety of interfaces. The customer's system may be used to manage sales, inventory, or other data collection information. The barcode scanners execute real-time embedded software. Software quality assurance (QA) efforts at Spectra-Physics use PC-based test systems with custom test beds for automated testing. The automated tests are scripts which execute on top of an internally developed application. Manual testing is executed using a duplicate of the customer's system or a simulator of the customer's system.

## 1.2 Product Life Cycle

At Spectra-Physics, products go through four main phases: concept, development, enhancement, and obsolescence. The New Products group owns products when they are in the concept and development phases. During those phases, New Products software development and QA personnel decide on product hardware, development tools, software test hardware, and software test tools. After a product survives extensive review and software QA testing, the product is available for general shipment. At this time responsibility for the product and test system is transferred to the CES group for ongoing enhancements and support.

**Figure 1: Product Flow**




## 1.3 Customer Engineering Support's Task

When the CES group receives a product from the New Products group, it receives a product package. A complete package includes documentation and tools to support the product hardware, software, and testing. The CES group uses this product package to accomplish its main task; to respond to customer requests for enhancements. The group also responds to customer complaints by fixing product problems. Whether the project is in response to a customer enhancement request or a customer complaint, the development and test process is the same. In this paper, a customer enhancement request refers to both enhancements and complaints.

The CES group consists of three software development and three software quality assurance (QA) engineers. Currently, the group is responsible for ten distinct barcode scanning products. Projects are generally staffed with one software development engineer and one software QA engineer. Enhancement project durations range from two weeks to six months, with an average length of about one month.

When we receive a customer enhancement request, we assign it a priority and schedule it as a separate project or combine it with a project that is already scheduled. Customer enhancements often require changes to the product software, which then requires enhancements to the automated and manual software tests. After the enhanced product is tested, appropriate changes are made in manufacturing for shipment.

Spectra-Physics Scanning Systems, Inc. is certified to the ISO9001 International Quality Systems Standard. This adds additional tasks for CES due to the ISO9001 requirement that processes be defined, followed, and documented. Activity logs are necessary so that periodic audits can verify compliance.

# 2. Improvement Process

## 2.1 Problem Areas in Testing

When our group was formed, we soon realized that we needed to make changes in order to efficiently manage the product enhancements and software testing. After a few brainstorming sessions we decided that we would realize the biggest gain if our whole group would concentrate its efforts on improving the testing process.

Our test systems caused a lot of frustration. We had four test systems, each unique in hardware and software. Each test system was set up for a specific product. This was inefficient because test systems often sat idle while a tester worked diligently on a product's one and only test system. Other test systems sat idle because there was no current project that used them. If we had test systems that worked for more than one product, we could improve efficiency. Test systems were also unstable. This often caused a tester to spend half of the scheduled test time fixing problems with the test system.

In addition to difficulties with test systems, none of the products had a complete set of test plans. Some of the products had no test plans at all. Most testing relied on the tester's knowledge of undocumented automated test scripts and the best way to perform manual testing. Our test coverage depended heavily on test scripts that had never been reviewed.

Project tasks were not well understood. This caused problems for predicting an accurate project completion date which led to poor customer satisfaction. This also led to ISO9001 audit problems since the project log did not have a standard set of tasks and was very hard to interpret.

Product quality in the field and the factory provided additional clues that there were problems with the test process. Two products with new features arrived at the new customer's sites and didn't work. The new features worked in the configuration that we tested but not the configuration that the customers were using.

We decided that the following test problems were the most important to fix:

- Project tasks were not understood
- Test systems were difficult to set up
- Tests were not performed consistently
- Test coverage was unknown
- Test results were not recorded

## 2.2 Improvement Process Steps

As we worked toward a better test process we advanced through each of the items listed below. Each step is key to a successful improvement effort.

- Define the improvement team charter.
- Diagram the current process.
- Gather and record improvement ideas.
- Assign priorities to the improvement ideas.
- Implement the improvements.
- Assess the results of the improvements
- Identify and implement additional improvements as needed.

To define the improvement team charter we needed to identify in general terms the areas that needed improvement. We decided to work on the areas of test consistency, reportability, and efficiency. Test consistency was important because it would lead to a more efficient and flexible process. It would also help improve the quality of our products. Test reportability was important because we had no traceability for software test or the problems that were found and resolved. Test efficiency was important due to the need for quicker turn around of product enhancements.

Next, when we tried to diagram the current process, we had a lot of difficulty determining exactly what the process was. Each person had their own method or interpretation of what they thought was the accepted method. There was no "standard" process. In the end, we did not define our current process. Instead, we defined the process that we wanted to follow.

Next, we gathered improvement ideas. We had brainstorming sessions. We talked with our customers in the factory, our customers in support engineering, and the New Product's test group. After all of our interviews and meetings, we had a list of about 20 improvement ideas. The ideas ranged from hiring a test librarian to writing a test policy. We prioritized the ideas by first asking our customers to prioritize them and then we prioritized the list as well. The final implementation list of priority ideas were a combination of our customer's top priorities plus our team's top priorities.

The longest phase of our improvement group's existence was the implementation phase. We implemented the top priority ideas plus a few additional ones that solidified the improved process. After an improvement was implemented, we judged its value and decided if the enhancement should be kept.

We succeeded in our improvement effort because of each team member's individual efforts. Our team consisted of a coach, a facilitator, a leader, and five regular members. The coach was our CES manager who attended most meetings and provided excellent support of our efforts. The facilitator was a process improvement expert from our Quality and Reliability department. He had experience with several groups who used the process improvement steps successfully. I was from CES and was the leader of the improvement effort. I worked to keep the meetings flowing smoothly and kept track of the team's current improvement projects and goals. The regular team members were from CES. They worked to define the process and make the improvements.

## 2.3 Improvement Strategies

During the improvement effort we implemented the following:

- Improved the management of tasks
- Modified our development procedure
- Updated the testing process flowchart
- Improved the test system configuration
- Incorporated test plan updating and reviews
- Started reviewing test scripts
- Started writing test reports
- Scheduled weekly review meetings
- Developed a problem tracking process

# 3. Improvement Implementation

This section describes the improvements that we implemented and explains why each was important.

## 3.1 Management of Tasks

Maintaining an accurate, up-to-date schedule is important to us and to our customers. Our customers want to know when their enhancement will be complete. Our group needs the schedule in order to plan and prioritize tasks aimed at meeting committed shipping dates.

A detailed schedule was one of the first improvements we implemented. Table 1 shows an example project schedule. The improved schedule also serves as a task list. Using the schedule as a task list has been an excellent tool for making sure that the process is being followed. Note: FSR/CCL stands for Field Sales Request/Customer Complaint Log which is being referred to as a customer enhancement request in this paper.

**Table 1: Sample Project Schedule**

| FSR/CCL # | Task Name | Dur. | Start | Finish | Original Commit | Owners |
|---|---|---|---|---|---|---|
| **F94-11-07-LD-01** | **SID-950LX** | **62d** | **11/7/94** | **2/9/95** | **2/15/95** | |
| | Define requirements | 2d | 11/7/94 | 11/8/94 | | |
| | Demo development | 6d | 12/19/95 | 1/4/95 | | Larry |
| | Demo testing | 3d | 1/5/95 | 1/9/95 | | Larry |
| | Demo available | 0d | 1/9/95 | 1/9/95 | | |
| | Customer validation | 10d | 1/10/95 | 1/23/95 | | |
| | Design review prep. | 1d | 1/24/95 | 1/24/95 | | Larry |
| | Design review | 0d | 1/24/95 | 1/24/95 | | |
| | Release development | 2d | 1/25/95 | 1/26/95 | | Larry |
| | Release testing | 4d | 1/27/95 | 2/1/95 | | Spencer |
| | Factory verification | 5d | 2/2/95 | 2/8/95 | | Jeff R., John H. |
| | Release | 1d | 2/9/95 | 2/9/95 | | Deanna |

# 3.2 Development Procedure

Figure 2 lists the principal steps of the CES software development process. The steps are an excerpt from the CES software development procedure. It states what is required for software development. We use this document and a collection of project outputs during ISO9001 audits to determine if the group is following the defined process. I have included this procedure because it shows how software testing fits in the CES process at step 7. We modified the development part of this document several times during the improvement effort. We found out that we had to define at least some of the development process details as we defined our test process, since the two processes rely on each other.

**Figure 2: CES Software Development Procedure**

| | Responsible Person | Activity |
|---|---|---|
| 1. | Sales force member, Customer Administrator, or other person | Completes Field Sales Request (FSR) defining input requirements. |
| 2. | Customer Support Specialist | Reviews FSR input requirements; clarifies any ambiguities with originator. |
| 3. | Applications Engineer | Develops demo software. |
| 4. | Customer Support Specialist | Sends demo software to customer, labeled as demo with FSR number. |
| 5. | Customer Support Specialist | Verifies demo software meets requirements via communication with customer or representative of customer. |
| 6. | Applications Engineer | Upon receipt of customer validation of demo, develops release software. |
| **7.** | **Test Engineer** | **Tests the released version of software and signs off for release.** |
| 8. | Factory Test Engineer | Tests the released version of software and signs off for release. |
| 9. | Applications Engineer | If software passes all tests, release software to the Documentation system. |
| If the activities cannot be carried out as described above, the Customer Engineering Support Manager authorizes alternative activities. | | |

# 3.3 Testing Process Flowchart

Each QA person in CES had come from a different development team and knew all about one product and its unique testing considerations. Each product had been developed and tested with a different degree of formality and under a different methodology. This was a problem since we wanted to be responsive to a customer's requests no matter what the product was or which tester was available to support the testing effort. To meet this need, each of us needed to be able to test any product. This meant we needed a uniform and consistent process which was flexible enough to support all of the products. Figure 3 is the process flowchart that we developed, and Figure 4 is a checklist that follows the steps in the flowchart.

Some of the key points of this process are: testing is planned for, test plans are written or updated, problems are tracked, risk is assessed, test logs are generated, and the process deliverables are reviewed. In addition to the flowchart and checklist, there is a guideline that gives a written description of the process.

**Figure 3: CES SW Testing Process Flowchart**

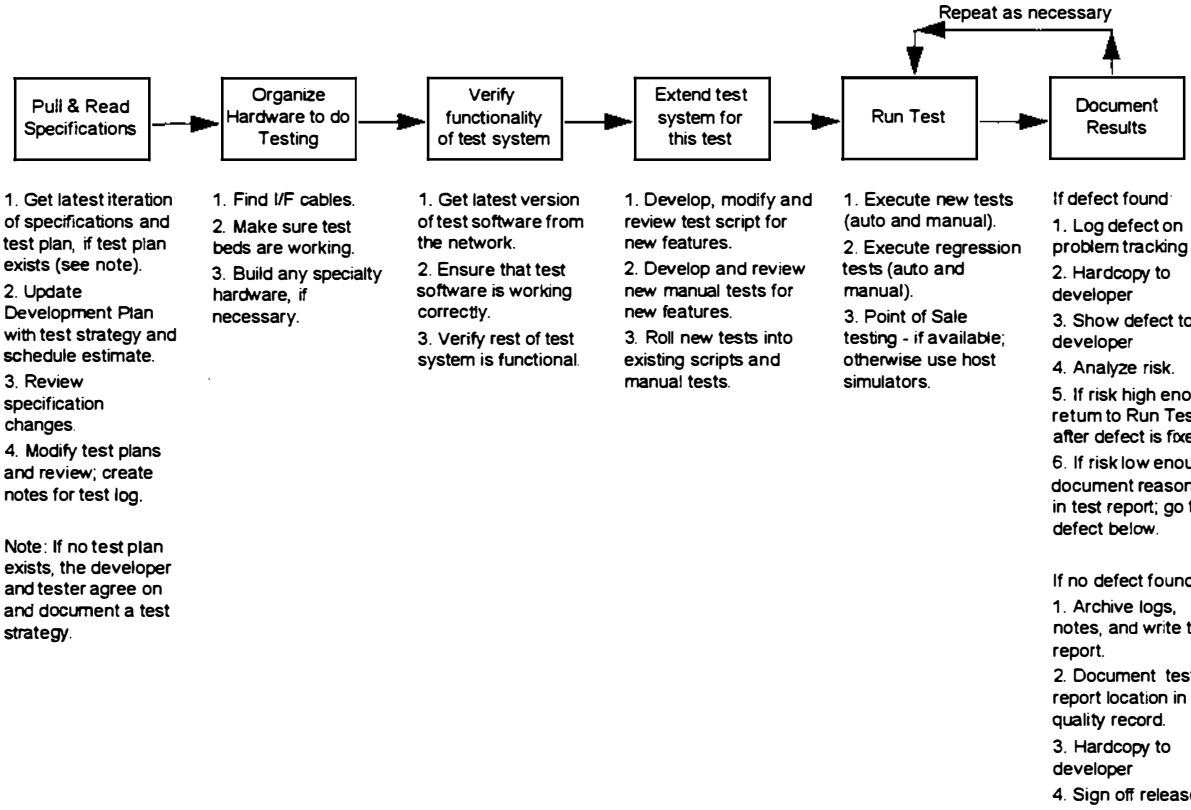



**Figure 4: CES SW Testing Process Checklist**

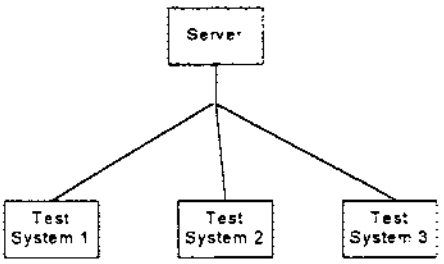
1. Get latest specifications and test plans
2. Obtain copy of customer enhancement request
3. Plan for project and estimate effort
4. Add test information and estimated effort to development plan
5. Review specification changes
6. Submit revised/new test plan for review
7. Setup test system and verify functionality
8. Write/modify automated test scripts
9. Submit test scripts for review
10. Perform automated/manual tests with new software
11. Record problems in problem tracking system
12. Analyze problem severity
13. Mitigate problems with management and development
14. Write test report
15. Archive test logs and test report
16. Place test report archive reference in quality record
17. Sign off release

# 3.4 Test System Configuration

When our group was formed, our test system configuration was found to have many problems. The test system software existed on individual PCs. Upgrading test system software to the latest revision required the use of many floppy disks. Upgrades happened very infrequently. When scripts were enhanced, they normally only existed on the machine that was used for the testing. Searches for the latest scripts involved looking on each test system. In addition, it frequently took half of the test time to set up the test system due to hardware problems.

The improvement team made great strides in the test system improvements. We connected each test system to a local area network. Now we could easily upgrade and save test software enhancements. We established a common directory structure on the network and installed the structure plus a uniform tool set on each test station. Now we could test any product on any test station. We upgraded the test system hardware so that it could support the growing requirements of incoming new products. We became more efficient and less frustrated.

**Figure 5: Test Systems Networked**




# 3.5 Test Plans

Before we defined our test process, we typically did not address the need for test plan development or extension. Paying little attention to test plans resulted in inconsistency, unknown coverage, and in some cases poor overall results. For each project we now carefully assess the need for and necessary extent of a test plan. We base our assessment on the nature of the project and the future of the product.

If the product is close to obsolescence, or if the enhancement is trivial, then we generate minimal test plan documentation. If our company plans a long future for the product, or if the enhancement is far-reaching or complex, then we put more work into the test plan. In all cases the group reviews and documents the test plans to note any problems or suggestions. At a minimum, we record a brief description of the test plan in the project's development plan. For more involved projects we update the existing test plan or write a new one if none exists.

# 3.6 Test Scripts

The majority of our testing is automated. No one has ever reviewed or documented most of the scripts that perform our automated tests. What could we do to verify that scripts test what they are supposed to test when they number in the thousands?

We talked about this issue many times and decided to start documenting the scripts that we modify or create. Figure 6 shows the test script template we developed. The template provides a format in which we document what the script tests and how it tests it. Now we review all script modifications as well as newly created scripts.

**Figure 6: Test Script Template**

```
;+++=============================================================
;+++.SCRIPT   Short Description
;+++=============================================================
SCRIPT FILENAME:
PURPOSE:
        Statement of this scripts purpose.
USAGE:
        Command line usage.
ARGS:
        Argument definitions.
DESCRIPTION:
        Detailed description of test approach, method, coverage, etc.
PDL:
        Program Design Language
ITEM METHODOLOGY:
        List of each requirement item that this test addresses, including a
        definition of "how" the item is satisfied.
PRECONDITIONS:
        Any conditions which must exist prior to execution.
EXECUTION KEYSTROKES:
        Definition of any "Hot Keys" that will abort, etc.
ERROR REPORTING:
        Method of error reporting, failure criteria, maximum
        errors until test abort, etc.
NOTES:
        Any special comments regarding the script.  Enhancements needed,
        considerations, etc.
REVISION HISTORY:
  Date              Author              Reason
-------------------------------------------------------------------
```

# 3.7 Test Reports

In the past, when software problems were resolved and the release was ready, the tester signed off the release and went on to the next project. Our group decided it would be a good practice to provide some traceability of the testing that occurred during the project. We consider it important to know and record which tests we changed, which tests we ran, what problems we found, and what we did about those problems. We developed the Test Report Template shown in Figure 7. Before we release the software, we generate a test report and make it available for future reference.

**Figure 7: Test Report Template**

PROJECT NAME:
PROJECT ID:
TESTER NAME:
DATE:
SOFTWARE ROM NUMBER:
TEST SYSTEM CONFIGURATION:
ENHANCEMENT LIST:
REFERENCE DOCUMENTS:
MODIFIED/NEW TEST PLANS:
MODIFIED/NEW SCRIPTS:
PROBLEMS FOUND AND RESOLUTION:
TEST SUMMARY:
    NEW FEATURE TESTS:
    TESTS TO VERIFY PROBLEM FIXES:
    AUTOMATED REGRESSION TESTS:
    MANUAL REGRESSION TESTS:

# 3.8 Weekly Review Meetings

Since we do many short projects and minor enhancements, our group frequently has to review numerous small changes to test scripts, software source code, and other documentation. Previously, each author called a review meeting as needed. Now we schedule a weekly team meeting to review all documents that have been completed by the team during the previous week.

We have benefited in many ways from the weekly review meetings. We keep meeting minutes and record *action items*, specific tasks assigned to individuals. Action items usually entail correcting documents according to the conclusions reached in the reviews. At the beginning of each meeting we go through all the open action items and check to see if they have been completed. When they are completed we close them. After the meeting, the list of open action items is updated and distributed to team members so that each team member has a current list of actions to work on. The minutes have provided an excellent way to assure that open action items are resolved, and as such have also provided the necessary project record for the ISO9001 audits. Lastly, we have eliminated practically all of the overhead associated with finding a conference room and allotting time for many small reviews.

During our meetings the team reviews the following items as they are completed, as well as occasional process improvements:

- Action Item Status and Closure
- Development Plans
- Test Plans
- Test Scripts
- Problem Closure
- Specifications
- Design
- Code

## 3.9 Problem Tracking Process

Our group had kept track of problems in our problem tracking system, but not consistently. Often a problem would be recorded, fixed, and verified but never closed. There were many problems in the system that remained as Open/Reported for years. We had no process for closing out problems.

We decided that we should identify old open problems when we write our project plans. Closing the problems would be part of the project and would be scheduled. We also recorded the following four step process for managing new problems:

1. Tester records problem with status Open/Reported
2. After fixing problem, developer changes status to Open/Fixed
3. After verifying fix, tester changes status to Open/Fixed/Confirmed
4. After closure review, developer changes status to Closed/Fixed/Confirmed

Note: Along this process, a problem may be classified as Closed/NoProblemFound or HoldWatchFor.

# 4. Improvement Results

## 4.1 Test Process

We found that a testing process defined by its users yields extra benefits. Most importantly, everyone on the team understands the process. We have fewer surprises because we have defined both our deliverables and the transfer of responsibility between software developer and software tester. We adhere to the process because it is ours and we continue to improve it. We verify adherence by regularly reviewing our project schedule and process deliverables.

Our process has given us greater flexibility. If we need extra test resources to meet customer commitments for a particular project, a team member who has relatively little experience with the product in question can work through the task by following the defined process. In this case, review of deliverables and the schedule provide a check for adherence.

## 4.2 QA/Development Communication

Communication between QA and development is key to a successful software development project. The process has defined some of the important communication paths. The defined communication occurs during reviews of development plans, specifications, test plans, code, and test scripts.

By defining the process we also define the responsibility for tasks. This has helped to speed up the entire process, since individuals are aware of their responsibilities and the sequence of events.

In addition to this, the group knows the process well since they developed it. Both development and QA understand what information is important to successfully test a product.

## 4.3 Test Coverage

Review of test plans and scripts has taken some of the "unknown" out of test coverage. Through reviews, QA and development are clear about what will be tested, and the final test report documents the results. We still have many test scripts that have not been reviewed, but the risk is always decreasing since the group is reviewing scripts as they are modified.

# 4.4 Test Reportability

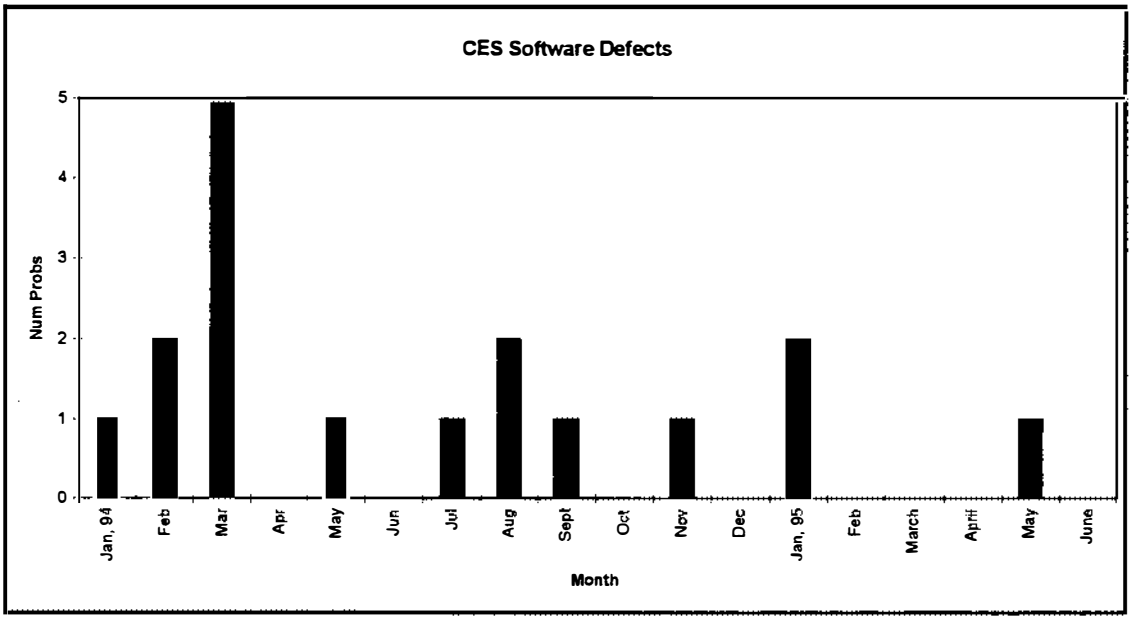
Test reports have been useful in two key areas. First, they have provided a clear record of the changes that were made in the testing area for a project. Second, they have provided a summary of the problems that were found and how those problems were closed. This second aspect has been helpful when the release decision is made. The test report gives an indication of the product's quality level.

# 4.5 Product Quality

Product quality is very hard to measure and more difficult to analyze. Figure 8 shows the number of software problems found by our customers in the factory and in the field since we began the process improvement effort. The data below only includes software problems for the products that our team originally supported. I did not include data from our new products since those problems do not originate from our efforts.

The graph shows a slight decrease in problems over the 16 months. The decrease may be due to many factors: maturity of the products, number of products sold, improvements in testing, improvements in development, and many others. Overall, this information will be more helpful in the future when we have more data and have a better view of the trends. Other forms of metrics would also provide a clearer picture. We may want to start collecting the number of problems that the CES QA group identifies and the amount of time that we spend on testing efforts.

**Figure 8: Customer Complaints and Ship Holds**




# 4.6 Test Efficiency

The efficiency of our testing has improved the most due to our defined process and our enhanced test system configuration. With the improved test system configuration, the testers are spending most of their time concentrating on the tests instead of problems with the test system. Since we defined the test process, all of our software changes have worked at the customer site at release time. By capturing problems before they get into the field our projects are more efficient.

## 4.7 Team Spirit and Cooperation

Team spirit is very hard to measure. We think that our team rates very high on all team spirit measurements. This is due in part to the improvement project that we worked on. We have many individual tasks in the support of product enhancements. The improvement project gave us a positive goal to work toward as a team.

Defining the test process also led to a common understanding of software QA. Developers and testers are more willing to work together since we understand the process as a whole

# 5. Conclusions

## 5.1 Consistency

Consistency was an important goal for our group because it would lead to a more efficient and flexible process. It would also help improve the quality of our products.

We obtained more consistency through several avenues. The definition of our process was a key accomplishment to obtaining more flexibility since the process defines a test process that covers all products. Test resources can easily be trained in the process and can be used in any test effort. The defined process also leads to efficiency since the task responsibilities and communication paths are defined. Our SW QA process flowchart and corresponding checklist describe the process we defined. They are shown in figure 3 and figure 4.

Writing test plans and developing our test script template were other avenues that we used for consistency. With the test plan and test script information documented, the testing can easily be reassigned to any test resource. This provides flexibility in resource allocation. The test documentation also increases efficiency since future projects can reuse the plans. The test documents are key to improving product quality. Test plans and scripts document the test coverage. Because we review these documents, improvements in coverage are implemented as the reviewers make improvement suggestions. Improved test coverage helps to find problems during the test effort and before the customers receive the product. Our test script template is shown in figure 6.

Our definition of the problem tracking process has led to an efficient path for problem resolution. The responsibilities are defined so everyone knows their role.

Adherence to these processes are maintained through weekly review meetings where action items are opened when they are identified and closed when they are completed. The improved schedule that we use provides another check for adherence since project steps are tracked and reviewed. A sample schedule is shown in table 1. Finally, ISO9001 audits verify adherence at a company level.

## 5.2 Reportability

Reportability was an important goal for our group because we had poor traceability for software test efforts or the problems that were found and resolved. Our test reportability comes through test reports and review minutes. Review minutes record action items that were identified in reviews of test plans, test scripts, and problem closure. Test reports provide the desired traceability of problem resolution, test coverage, and test changes. Our test report template is shown in figure 7.

## 5.3 Efficiency

Efficiency was important due to the need for quicker turn around of product enhancements. We have improved the efficiency of our testing through the definition of our process as discussed in the Consistency section above. In addition to the process definition, we have made strides toward greater efficiency through our test system improvements. Having an organized, consistent test environment helps to minimize the inherent problems with custom test systems that are normally required to test products with embedded software.

## 5.4 Improvement Process

We successfully implemented many improvements to our software QA test process and environment. This success is due to our process improvement expert, the individuals on the team, and to the defined improvement process that we used. The improvement process kept us on a path to success.

## 5.5 Future Plans

Our improvement effort addressed many of the important issues with our SW QA test process. There are many other items on our list of possible improvements. One item of further work in the test area is measuring our test process with additional metrics. Another item is defining the requirements for a transfer of software test tools from the New Products department to our group. These items are on our list of improvement ideas but we will be concentrating on the definition of our development process next. Our development process has improved since we started the test process improvement effort but it needs to be defined in greater detail and documented. We have decided to spend 2 hours a week on improvement efforts in this direction.

# PARSE: Problem Analysis and Process Management for Software Maintenance

Dr. Robert Bruce Kelsey

Digital Equipment Corporation
301 Rockrimmon Blvd., South - CX02-1/8I
Colorado Springs, CO 80919-2398
email: kelsey@cookie.enet.dec.com

ABSTRACT:

PARSE, Problem Analysis matRices for Sustaining Engineering, uses a detailed analysis of the software problem report as a means to motivate and control software maintenance activities. It combines the techniques of causal analysis with product-specific and release-specific analysis matrices in support of release management, standardizes prioritization and risk assessment of remedial work, and provides defect data necessary to meet continuous improvement goals.

BIOGRAPHICAL SKETCH:

Dr. Kelsey's experience in software quality assurance includes three years in maintainability engineering for both operating system software and storage hardware subsystems and five years as a principal software engineer and technical project leader in Digital Equipment Corporation's OpenVMS™ and Storage Management Software engineering groups. He currently directs the Quality Initiative program in the Storage Management Software Engineering group, performing internal audits and providing process improvement support for 20 product teams. His publications include poetry, essays in philosophy and amateur astronomy, articles on software maintenance project management in the *Quality Newsletter* and (forthcoming) an article applying philosophical linguistics to problem diagnosis in *Software Engineering Notes*. He received his cross-disciplinary doctorate from the University of Iowa in 1982.

## INTRODUCTION

Software maintenance is an unsavory business, full of "bugs," "crashes," "dumps," "hangs," and "corruption." Maintainers lunge from one point fix to another, schedules totter precariously at the edge of extinction, project leaders develop an array of nervous disorders, and the backlog of problem reports grows, and grows, and grows. How can you prioritize, schedule, and complete maintenance tasks when the only thing you know for certain is that the worst bugs will show up at the last minute? How can you improve the quality of your maintenance process when the *modus operandi* is "problem received, fix applied, problem received, fix applied" etc.?

In 1991, the I/O subsystem engineering team in Digital Equipment Corporation's OpenVMS group had to find answers to those questions, quickly. The long range plan called for four new releases of the I/O subsystem in the next three years, two releases to support the VAX hardware platform and two to migrate OpenVMS onto the AXP platform. Most of the engineers were assigned to the extremely aggressive AXP port, leaving few resources to maintain and improve the code base. To make matters worse, the I/O subsystem was a collection of aging device and network drivers, easily perturbed by changes elsewhere in the operating system. And we were required to support far more configurations than we had at our disposal for debug and unit test.

We knew we had to find a way to manage maintenance, not just respond to problem reports. We had to find a way to prioritize problems, taking into account not only their impact to the customer base but also their impact to the multiple, concurrent release schedules. We had to stop fixing bugs and instead start looking at the stability and maturity of the I/O subsystem as a whole, applying limited resources where they would most benefit the entire customer base.

PARSE, Problem Analysis matRices for Sustaining Engineering, was developed during that first hectic year in OpenVMS. It started as little more than a checklist of defect types roughly analogous to a formal inspection checklist. We quickly found that such a checklist provided visibility into the root cause of the defect being reported,[1] but it did not help us prioritize our maintenance tasks. We eventually developed a set of prioritization matrices that took into consideration characteristics of the problem report as well as characteristics of the defect. Those matrices helped us standardize problem analysis, they helped us prioritize our maintenance tasks and they helped us assess the risks of proposed fixes. When several years later a similar matrix scheme was adopted by the Storage Management Software engineering group, we discovered that PARSE techniques could be used to assess release readiness and product maturity. We even used PARSE to successfully predict the types of defects that would be reported from the customer base after the product's release.

The first half of this paper introduces the PARSE matrices, showing how we populated them and how those matrices helped us prioritize problem analysis and corrective maintenance. In the second half, we'll switch perspective from reactive to proactive, and examine how PARSE has been used to assess product maturity and identify quality goals of preventative maintenance.

## DETERMINING THE PRIORITY OF THE PROBLEM REPORT

In the sense used in this paper, software maintenance is corrective and complaint driven. Unfortunately, problem reports are seldom informative and never predictable. Customer complaints range from the meek and pitiful "I think my system crashed" to 14 screens of crash dump analysis data from an overzealous system manager that is totally useless because the problem lies elsewhere. And of course, most problem reports are showstoppers, if not to engineering then at least to the person who filed the report. In the worst case, the maintainer receives via email or some other mechanism an unintelligible one line complaint, and before the maintainer has had time to

acknowledge its receipt, the Vice President for the division already has left voice mail wondering when the fix will be ready for this 2 billion dollar account.

Visibility of the problem too often drives maintenance. No one would argue that a report from such a customer should be ignored. But it *is* reasonable to ask "When *should* it be addressed?" Are there other, more potentially damaging defects at large in the customer base that warrant attention first? And visibility can be insidious in distracting maintainers from the actual technical issues. In OpenVMS, maintainers had for years responded to any and all problem reports coming in from one specific test site. When we first set about creating PARSE matrices, we knew from our own experience that most of the problems reported from this test site had never, would never, occur in the installed base. But we had nothing to support our beliefs.

We needed a way to determine the importance of the problem report (as distinct from the incidence of the defect that caused it), just so we knew which problem reports to address first. We also needed a way to determine how much of the installed customer base might reasonably experience this problem. And we needed a way to determine how often a customer might encounter the defect underlying the problem. If we knew a problem's frequency of appearance, we could differentiate between a politically sensitive customer complaint and an inexcusable technical blunder. If we knew a defect's penetration in the installed base as well as its frequency of appearance, we could politely acknowledge the test site's report and move on to fixing bugs that actually put the customer base at risk. The first three PARSE matrices we created focused on the problem, not the defect, and prioritized it based on Visibility, Penetration, and Frequency.

Matrix 1: Visibility

| Type | Priority |
|---|---|
| Critical customer outage, systems down | 6 |
| Problem report against released version | 5 |
| Problem preventing another development group from meeting field test deadline | 4 |
| Problem report from prominent customer field test site | 3 |
| Problem report from test facility, field test code | 2 |
| Problem report from customer site, field test code | 1 |

Matrix 1 is a sample of the Visibility matrix. The problem report characteristics we chose, and the priorities we assigned them, were based solely on our particular working environment. But the matrix did try to strike a balance between technical issues and problem management issues. Visibility allowed us to put limited resources to work on those problem reports that held the highest risk, first to the installed base, second to the engineering organization in general, and third to the current field test participants.

It often happened that a report came from a source warranting immediate attention, but the problem described in that report did not warrant immediate remedial engineering. After an engineer made a first pass over the problem report, the next task was to determine the penetration of the problem.

Matrix 2 shows a sample matrix for Penetration into the customer base. At this stage in the analysis, no root cause was known, and so we settled for coarse differentia - released versions, known conditions, etc. We used the Penetration matrix to help specify the actual, versus perceived, severity of the problem. The High, Medium, and Low labels used in the reporting system lacked sufficient granularity to help us assess actual priority. What constituted a "critical" problem to one maintainer, or one customer, was often not a critical problem for another maintainer or customer. And we wanted to avoid the situation where a problem report from one highly visible, irascible customer who hadn't upgraded to reasonable hardware revision levels deflected our attention from deficiencies in *product* functions.

Matrix 2: Penetration

| Type | Priority |
|---|---|
| Problem possibly affects all current customers | 5 |
| Problem probably occurs in a specific release | 4 |
| Problem possible only in this baselevel | 3 |
| Problem manifests in a specific software configuration | 2 |
| Problem manifests in a specific hardware configuration | 1 |

Comparing Penetration with Visibility prevents the unique problem report from being ignored in favor of its more Visible peers. Visibility tracks problem report origin; Penetration supplements this by revealing the 'demographics' of the problem itself. This assists release management because scheduling and resource allocation can be directed to those problems that pose a significant risk to the whole installed base, rather than those that 'generate the most heat in the system.'

But we soon found that penetration in the installed base was not sufficient to prioritize the problem report. Most user-initiated operations involved our product in one way or another. Hundreds, perhaps thousands, of lines of code might be executed, in innumerable combinations, before just the right, or wrong, combination produced an error. We needed some sense of the frequency with which this sequence of events might occur, regardless of how many sites had already installed the latent defect.

Matrix 3: Frequency

| Type | Priority |
|---|---|
| No identifiable precursive events | 5 |
| Specific precursive event sequence on local system | 4 |
| Specific precursive event sequence distributed across multiple systems | 3 |
| Specific hardware/software interface | 2 |
| Not known/unreproducible | 1 |

Matrix 3 shows a simplified version of a Frequency matrix. (The matrix we actually used was based on driver specific interfaces and VMScluster distributed synchronization techniques and is more detailed than is necessary for this discussion.) Frequency in the sense used here is not the incidence of the problem report but instead reflects the potential incidence of the defect's symptom(s). Thus a problem report that is generated during "normal" product functions would have a higher priority than one which requires a specific hardware/software interface for its manifestation. A problem that manifests itself in local system operation is reasonably higher priority than one that occurs in a distributed environment, on the assumption that if the code is defective in local functions it is likely to be defective in widely distributed functions.

## Determining the Priority and Risk of the Fix

To determine the priority of the problem report, we simply summed the assigned priorities. Those problem reports with the highest priorities were assigned to developers who then performed more detailed analysis of the problem. The developer determined the nature of the defect underlying the problem report and used the PARSE matrices to prioritize its remedy. The matrices used in this stage were populated after a thorough examination of the following product characteristics:

1) The product functional requirements and the code paths that implement these functions.

2) The failure characteristics of the product and the effects these failures have both on discrete operations and on the system environment.
3) The architecture(s) and interface(s) within which the product must function.[2]

Individual entries in each matrix were assigned a priority, based on one of the following:

1) The original release goals of the product.
2) The reliability of both the product and of previous point fixes
3) The goals of the current release

We found that we needed flexibility in prioritization for two reasons:

1. Spoilage - Portions of the I/O subsystem were being used in environments for which they were not designed, and we needed to view the product characteristic from the current customer's perspective, not from the designer's perspective. (Try telling Lloyd's of London that you can't stop the periodic crashes of their new tape silo because the tape driver was never designed to handle the I/O load.)
2. Release stability - Operating system releases were driven by the need to release new functionality in areas other than the I/O subsystem, so we had to accommodate I/O subsystem maintenance to external qualification and release schedules. We needed a process by which we could assess the risk of a fix from several perspectives at once:
    - the risk to the current release if the fix turned out to be complex to implement and difficult to qualify
    - the risk to the customer base if we did not fix the problem in this release
    - the risk to the overall stability of the I/O subsystem in the event that correcting one defect stimulated others it had previously masked

The matrices that follow are for the purpose of illustration only, and are more generic than the actual, product-specific matrices we used.

Matrix 4: Operational Dependence or Context

| Type | Priority |
|---|---|
| Normal user-initiated operation | 7 |
| User-initiated error recovery operation | 6 |
| Product-specific internal error recovery on a control operation fails | 5 |
| Product-specific internal error recovery on a data operation fails | 4 |
| Multi-thread synchronization fails | 3 |
| System interface call fails | 2 |
| Product installation or restart | 1 |

Operational Dependence or Context is the fourth problem characteristic we PARSEd. The example in Matrix 4 illustrates the kinds of entries that might be included in this matrix. In actual use, the matrix would include all the functions and dependencies for the product. This particular example is clearly skewed towards availability and reliability, and assumes a functionally mature user interface. One could certainly prioritize the exact same characteristics differently under different conditions. There are certainly adaptive maintenance situations where performance considerations would be as high a priority as certain types of error recovery. And one might assume that failures to install a product would be high priority - if you can't install it, it isn't much good to the customer - but since we have a mature and reasonably tested product here, we can assume that installation failures are special cases.

Considered in conjunction with Operational Dependence or Context, the Product or System Effect matrix shown in Matrix 5 helps identify the impact on product availability and reliability. Failure in a system interface call may not

be high priority for the product seen in isolation. But if that failure results in a system crash, obviously the defect needs to be found and fixed before, say, one investigates an incorrect status display. In this sample matrix there is yet another example of how release goals can be factored into the prioritization of the matrix entries. The priorities of local and distributed operations have been swapped here compared to the Frequency matrix on the assumption that a failure in a distributed operation impacts more users than one that occurs locally. That assumption might *not* be appropriate for a network protocol failure in which an alternate server can be found. In this case the matrix should of course be modified to include entries which clearly distinguish recoverable from unrecoverable effects.

Matrix 5: Product or System Effect

| Type | Priority |
|---|---|
| Data corruption | 6 |
| System failure | 5 |
| Distributed operation failure | 4 |
| Local operation failure | 3 |
| Incorrect status displayed | 2 |
| Unacceptable performance | 1 |

We discovered that at this stage in the PARSE analysis, even before we had identified the root cause of the problem, we were often in a position to assess the risk to the customer base of any fix we might provide. A problem that appears to have high penetration, in a function frequently used, with deleterious effects on the customer environment, has greater liability to the maintenance team and the corporation if the fix is a "bad fix." And we also used PARSE analysis in real-time to modify test requirements and to assess qualification schedule impact. A fix to a problem in distributed operations that currently results in system crashes has higher risk potential, and will require more testing time and resources, than will a fix to a performance deficiency that affects only a specific hardware platform.

Of course, we also PARSEd the problem report for its root cause. We used two matrices, Defect Type and Defect Location. The Type matrix resembled the defect analysis checklists found in the most software project management texts[3,4,5,6,7,8,9] except that we added several categories specific to our development environment. An abbreviated example is shown in Matrix 6. We used this matrix in several ways. For example, the data collected from this matrix helped us confirm our belief that the lack of an interface specification was a primary reason for defects introduced during development. And we of course used trends in the defect type data to focus code reviews and inspections. We rated Types according to both the ease of defect removal and the risk of defect removal. Developers code typos, that's a fact of life. Fortunately, typos are easy to recognize, and seldom require re-qualification of the product. Coding errors therefore had lower process priority than state bit manipulation errors, which often do require a thorough understanding of the product, careful review, and re-qualification testing to prevent bad fixes.

Matrix 6: Defect Type

| Defect | Process Priority |
|---|---|
| Control structure state bit - state transition violation | 4 |
| Context not saved around Fork & Wait | 3 |
| Lock value block not initialized | 2 |
| Coding - syntax | 1 |

The Defect Location matrix was populated from an analysis of both functions and error paths in the product. The priority of each matrix entry was based on the previous PARSE analyses. An extract from our matrix is shown in Matrix 7.

Matrix 7: Defect Location

| Defect | Priority |
|---|---|
| Distributed fault management - synchronization module X, device Y | 17 |
| Local buffer management - performance assists | 13 |
| Distributed fault management - retry logic for error case Z | 10 |

In our particular situation, we knew that certain paths through the fault management code were problematic. Fixes to a particular synchronization module often perturbed other segments of the code. And there was a specific class of device errors which were not implemented consistently across all device types, and so any fix for one device might debilitate error recovery for another. The performance assists had been developed exclusively by a developer who was no longer in the company, and who had a taste for performing gymnastics on the interrupt stack with fork and wait routines.

The Priority values in this example may appear counter-intuitive. One would assume that a defect in complex code in an execution context guaranteed to crash a system on the slightest error is higher priority than a defect in a distributed fault handling algorithm for a particular device. And if one considers only the systemic effect of a defect, it certainly is. But the PARSE matrices were intended to provide defect profiles. They were intended to help put point fixes into perspective. So the Priority value was determined by previous PARSE data. Example 1 shows how the synchronization and performance assists locations compared with one another.

Example 1:

| Location | Penetration | Frequency | Operation | Effect | Total |
|---|---|---|---|---|---|
| Synchronization | 4 - all customers | 3 - distributed event sequence | 5 - internal error, control operation | 4 - distributed operation failure | 17 |
| Assists | 2 - specific s/w configuration | 2 - specific h/w - s/w interface | 4 - internal error, data operation | 5 - system failure | 13 |

It is certainly possible to disagree with the categorization in Example 1. The important point is that PARSE provided us a disciplined approach to setting priorities and determining risk. Instead of "problem received, fix applied," we had an explicit, and modifiable process for analyzing problem reports and determining the priority and risk of a fix. We no longer had to try to make tradeoffs between fixing one "High" priority problem report and another, with indeterminate results. If we checked in a fix for the Synchronization problem, we knew how much of the customer base would receive the benefits of that fix. If we had to neglect a synchronization fix and instead fix buffer management in the performance assists, we knew the liabilities of that release for the customer base.

## Release Readiness, Product Maturity, and Continuous Improvement

Releases full of seemingly unrelated point fixes seldom satisfy either customers or engineering program managers. A maintenance *strategy* is essential both for continuous improvement of a product and for maintaining customer satisfaction. Viewed as a problem 'profile,' the PARSE matrices allow the product team to assess the problem report from the perspective of product stability and maturity, they provide a method for "reading the tea leaves" [3] that does not require clairvoyance. This is crucial for a *long-term* maintenance strategy: trends in the characteristics of problem reports indicate areas requiring special attention *now*, either to avoid high problem report incidence later or to forestall certain types of problem reports that interfere with product goals.[10] In this respect, PARSE is also a means to identify both process deficiencies and *preventative* maintenance requirements that should be part of any continuous "product" improvement initiative.

Clearly, if 60% of the problems reports for the product in a given release all occur in the interfaces between modules *within* the product, the product is neither mature nor stable. Why it is unstable is best understood by examining the Defect Types that occur in that 60% segment of the problem reports. If the majority of those Types are coding, the product is probably unstable because it was released before it was properly qualified.

Extrapolating from this information, the engineering group would probably initiate a process improvement requiring more code inspections and more specification based testing of the product as *preventative* maintenance. The collected data from both the defect matrices and the Operational Context and Penetration matrices can help a project leader justify this up-front expense: if coding errors cause faults in normal operations in a majority of the customer base, future maintenance expenses are likely to be extremely high. Such information may be useful not only when negotiating next year's engineering budget - it may also help appease the customer whose problem was not fixed in a particular release because the team chose to fix other problems that actually put that customer at greater risk.

Let me offer another example of how PARSE can be used to determine product stability and preventative maintenance priorities. In 1994, I joined the RAID software development team in the Storage Management Software group, four months before the scheduled release of their second version. They were adding major new functionality to an application layered between the user QIO interface and the I/O subsystem. And they were trying to qualify that version against a new release of the operating system. They were using an electronic notes conference for problem reporting, they prioritized problem reports as high, medium, and low, and the specifications were 1 year out of date.

According to the project plan, the group was actually three weeks ahead of schedule. A cursory look at the problem history and a quick adaptation of the PARSE matrices to the new product showed that that would soon not be the case. Figures 1 and 2 show the problem report breakdown against abbreviated forms of the Operational Context and the Frequency matrices we used (thus they differ from the more generic matrices discussed previously). Problem report characteristics are listed in descending priority from highest to lowest.

Figure 1:

| **Frequency** | % of reported problems |
|---|---|
| No unusual context, event, or command | 45 |
| Specific local event sequence, specific software sequence | 25 |
| Specific cluster (distributed) event sequence | 20 |
| Specific hardware/software interface event | 10 |

Figure 2:

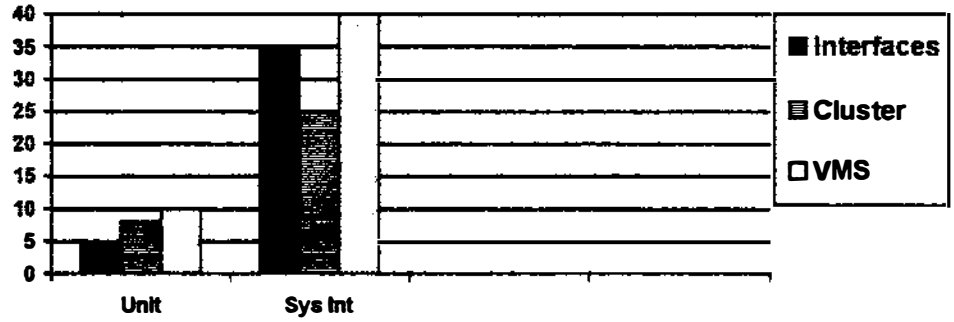
| **Operation Dependence or Context** | % of reported problems |
|---|---|
| Normal: Read, write, DCL commands | 60 |
| RAID internal error recovery on data path fails | 0 |
| User member mgt operation fails because RAID doesn't recover from an error in its own functions | 15 |
| RAID member mgt operation fails because RAID cannot recover from an error in its own functions | 0 |
| New node, local init, local config | 10 |
| RAID operation fails because it cannot recover from a VMS event/failure | 25 |

The Frequency matrix showed that most of the problems reported from the test group were in normal operations, and the Operational Context matrix showed that these appeared in critical (normal) functional paths and in the RAID-OpenVMS interface. In effect, the code was still in the functional testing phase. But the limited incidence of reported problems in internal error handling was most disturbing: defects in functions had prevented the test group from performing significant qualification of the error handling algorithms.

*Development* might have been three weeks ahead of schedule, but it was clear that the product could not be fully qualified in time to meet the committed ship date. The PARSE data was persuasive enough to motivate increased discipline in unit testing and to obtain additional resources for more aggressive functional testing. By the time the product was ready for field test, the product functions were fully tested.

But were we ready to start that field test?. PARSE data suggested the answer was no. An analysis of the Defect Location data showed that the majority of the defects recently found lay in the RAID-specific error recovery interfaces and in the RAID software's ability to recover from failures in the OpenVMS I/O subsystem beneath it (Figure 3).

Figure 3:




Containment was low in unit testing because we had been successful in removing those defects that could occur in the limited configurations available for unit testing. But in the more complex and distributed environment of systems integration testing, the RAID software simply was not robust enough for release to the customer test sites. Field test was delayed while we modified the systems integration test environment to focus specifically on testing RAID software responses to OpenVMS induced failure scenarios.

PARSE proved indispensable throughout the release cycle in helping us assess our progress against the product goals and in helping us assess the stability of the product. The product passed the final acceptance tests. Engineering management was happy. Product management was happy. The developers were suspicious. In the last month of product development, we had seen problem reports that PARSEd as low priority according to the release goals on all matrices. But they had one thing in common: their Defect Type was always an "Unanticipated distributed synchronization algorithm." In plain English, there was good reason to believe the product would work correctly on smaller VMSclusters but not on larger ones. *Before* the first distribution media shipped from Digital, and *before* the project plans for the next release had even been discussed, we were working on changing product initialization and streamlining the product's two-phase commit protocols. No one wants to get problem reports from the field; but in the next six months, the development team received only 3 reports, all of them identifying scalability issues, and in all cases we either already had the fix in hand or were testing it.

## SUMMARY & FUTURES

Software maintenance by its nature is the archeology of minutiae - the phrase "point fixes" is entirely apropos. While it requires intuition, imagination, technical expertise, and a dash of perverse cunning, it also has to be manageable, it must have procedures that can be standardized, evaluated, and modified as necessary.

In our experience, Problem Analysis matRices for Sustaining Engineering has the following benefits:

1. It provides a standard approach to interpreting and prioritizing problem reports, regardless of release goals, staff changes, or problem report volume
2. It assists real-time release planning (scheduling, populating a release, allocating resources) because priorities are explicit and new problem reports are subsumed within a long range, preventative maintenance plan.
3. It assists continuous improvement efforts by providing an explicit method for analysis and prioritization that can be examined, measured against release goals and staffing resources, and modified accordingly.
4. Should release goals or priorities change the matrices can be adjusted if necessary without disturbing the *discipline* of analysis used in future problem analysis.

PARSE analysis focuses attention on the critical aspects of software project management: the conformance of the product to release goals and the significance of a problem report for the product and the entire installed base. OpenVMS management has recently mandated the use of PARSE analysis for problem report prioritization throughout the entire engineering organization. In Storage Management Software, we have recently begun a quality initiative intended, in part, to use PARSE techniques to standardize project management across our entire product set. And as the storage group looks ahead to possible ISO 9001 certification, it is comforting to know that PARSE based analysis has brought us that much closer to meeting some of the procedural requirements for maintenance specified in the ISO 9000-3 standard.

## REFERENCES


1. P. F. Wilson, L. D. Dell, and G. F. Anderson. *Root Cause Analysis: A Tool for Total Quality Management.* ASQC Quality Press, 1993.

2. R. B. Kelsey. How To Assess and Improve the Quality of Both Software Product and Process. *Quality Newsletter*, **1**, (4) 9-10 (1994).

3. R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice Hall, 1992.

4. J. Martin and C. McClure. *Software Maintenance: The Problem and Its Solutions.* Prentice Hall, 1983.

5. C. McClure. *Managing Software Development and Maintenance.* Van Nostrand Reinhold, 1981.

6. M. W. Evans and J. T. Marciniak. *Software Quality Assurance and Management.* John Wiley & Sons, 1987.

7. W. S. Humphrey. *Managing the Software Process.* Addison-Wesley, 1989.

8. W. E. Perry. *Managing Systems Maintenance.* QED Information Services, 1981.

9. R. B. Grady and D. Caswel. *Software Metrics.* Prentice Hall, 1987.

10. J. S. Collofello and B. P. Gosalia. An Application of Causal Analysis to the Software Maintenance Process. *Software - Practice and Experience*, **23**, (10) 1095-1105 (1993).

# SELECTION OF A DEFECT TRACKING SYSTEM

**Peter Hantos,**
CR&T/CEC/SCC
Xerox

**Raymond T. Li,**
CR&T/ADSTC/ADSS
Xerox

**Sally Robinson,**
CR&T/ADSTC/ADSS
Xerox

## ABSTRACT

In order to efficiently develop and deliver software, the development organization must possess a powerful defect tracking system for logging and storing software defect information. The required services include the examination, editing and querying of previously submitted defect forms, and the generation of management reports. The right tool should also properly guide and facilitate the overall defect tracking process.

During the selection we addressed both strategic and tactical issues. Strategically we needed to find a tool which had all the required features and was going to serve the organization for a long time, but also satisfied our tactical requirements by being available immediately and having the right features to facilitate legacy data migration from the old system.

It would be certainly nice to standardize all tools on the corporate level, but unfortunately this unified approach is not always feasible. Also, an important part of the process is to understand and satisfy the actual customer requirements. For this reason the presentation concentrates on recommended selection practices instead of a final tool recommendation. The two-tiered approach we describe in the paper is a sensible method to get fast and realistic results.The main idea of this approach is to define the vital few criteria to narrow down the list of available tools, and execute a thorough benchmark of those tools only which passed pre-selection.

- * -

**Peter Hantos, PhD** is currently Principal Scientist in the Corporate Engineering Center. His tasks are coaching and mentoring process improvement teams across Xerox, identifying and sharing best practices, methods and tools, establishing repositories of guidelines, examples, and templates, facilitating the acquisition and development of software process training. In his previous position as Manager of Software Engineering & Tools he was leading various tools benchmarking efforts, and he was in charge of the defect tracking system selection task-force.

**Raymond T. Li** and **Sally Robinson**, as members of the Software Engineering & Tools group played key roles in the benchmarking process. They are currently providing tools and methodology support for three major organizations in Xerox Corporate Research & Technology.

**Corresponding Author:**

Peter Hantos
Xerox
Mailstop ESAE-375
701 South Aviation Blvd.,
El Segundo CA 90245
Phone: (310) 333 - 9038
Internet: phantos.es_ae@xerox.com

**PRESENTATION OUTLINE**

1. Introduction
2. Sample defect tracking process
3. How to define a defect tracking process?
4. The two-tier process to select a defect tracking tool
5. Tool selection strategies
6. The initial screening process
7. Detailed evaluation
8. Licensing issues, cost comparison
9. Concluding remarks

Appendix

## 1. INTRODUCTION

The authors of this presentation were chartered with the task of selecting a defect tracking system for one of the Xerox organizations. As part of that effort we also analyzed the tool selection processes in different Xerox divisions. We found the common problem was that most of the time they started with an excessive list of requirements, trying to be all inclusive. As a result, the selection process became extremely laborious and long, and still, most of the time the principals were unable to decide on an appropriate 3rd party tool or get commitment for an in-house development. We believe that the two-tier approach, used by our team is the key to get fast and realistic results. The main idea of the two-tier approach is to define the vital few criteria to narrow down the list of available tools, and execute a thorough benchmark of those tools only which passed pre-selection.

In order to develop and deliver reliable software modules, the software development organization must possess a powerful defect tracking system. Robust defect tracking process is specified as a Level 2 activity in the Software Engineering Institute's Capability Maturity Model (SEI/CMM). One of the Level 2 KPAs (Key Process Areas) of the CMM model is Software Configuration Management, and the description of Activity 5 reads as follows: "*Change requests and problem reports for all configuration items/units are initiated, recorded, reviewed, approved, and tracked according to a documented procedure*". **[1]**. The CMM does not mandate the utilization of any particular tool, in fact manual processes are acceptable as well, as long as they are well defined, documented and followed. Nevertheless, for larger organizations, working on complex products, the manual approach would be inadequate. It would be also nice to standardize all tools on the corporate level. Unfortunately due to platform and development environment differences, or the presence of legacy systems and databases this unified approach is not always feasible. For these reasons the authors are presenting recommended tool selection practices instead of final tool recommendations.

Since the presentation is the result of an actual experience, we have to explain some constraints of the selection process, and as a result, constraints for the paper itself. This particular organization already adopted a fairly mature defect tracking process, so the idea itself did not have to be "sold". This was the good news. The bad news was that they were using a very ineffective, proprietary legacy system, and in that database accumulated over 3000 defect reports (Called ARs or Action Requests in Xerox lingo) already, which needed to be saved and migrated to the new system. The tool was going to be selected for a software development organization, where defect reporting via phone was not acceptable. Consequently the defect tracking and reporting functionality was more important than the need for customer-support call tracking (e.g. help-desk or hotline type operations). In addition, the computing environment consisted of primarily Sun/SPARC workstations running Unix (SunOS 4.1.x and 5.x, to be exact). Due to various corporate initiatives, i486-based PC's made their way also to a certain segment of the population. The use of Apple Mac's or other processors was definitely very low at the time of the benchmark.

---

[1] Key Practices of the Capability Maturity Model, Version 1.1 by M.C. Paulk, C. V. Weber, S. M.Garcia, M. Chrissis, M. Bush., CMU/SEI-93-TR-25, ESC-TR-93-178, February 1993.

## 2. SAMPLE DEFECT TRACKING PROCESS

The very first error somebody can make is rushing into the selection of a tool, withouth understanding the process. In the absence of process, it should be the first step of defining an appropriate one for the organization. We will present a simplified, sample defect tracking process, for illustration purposes.

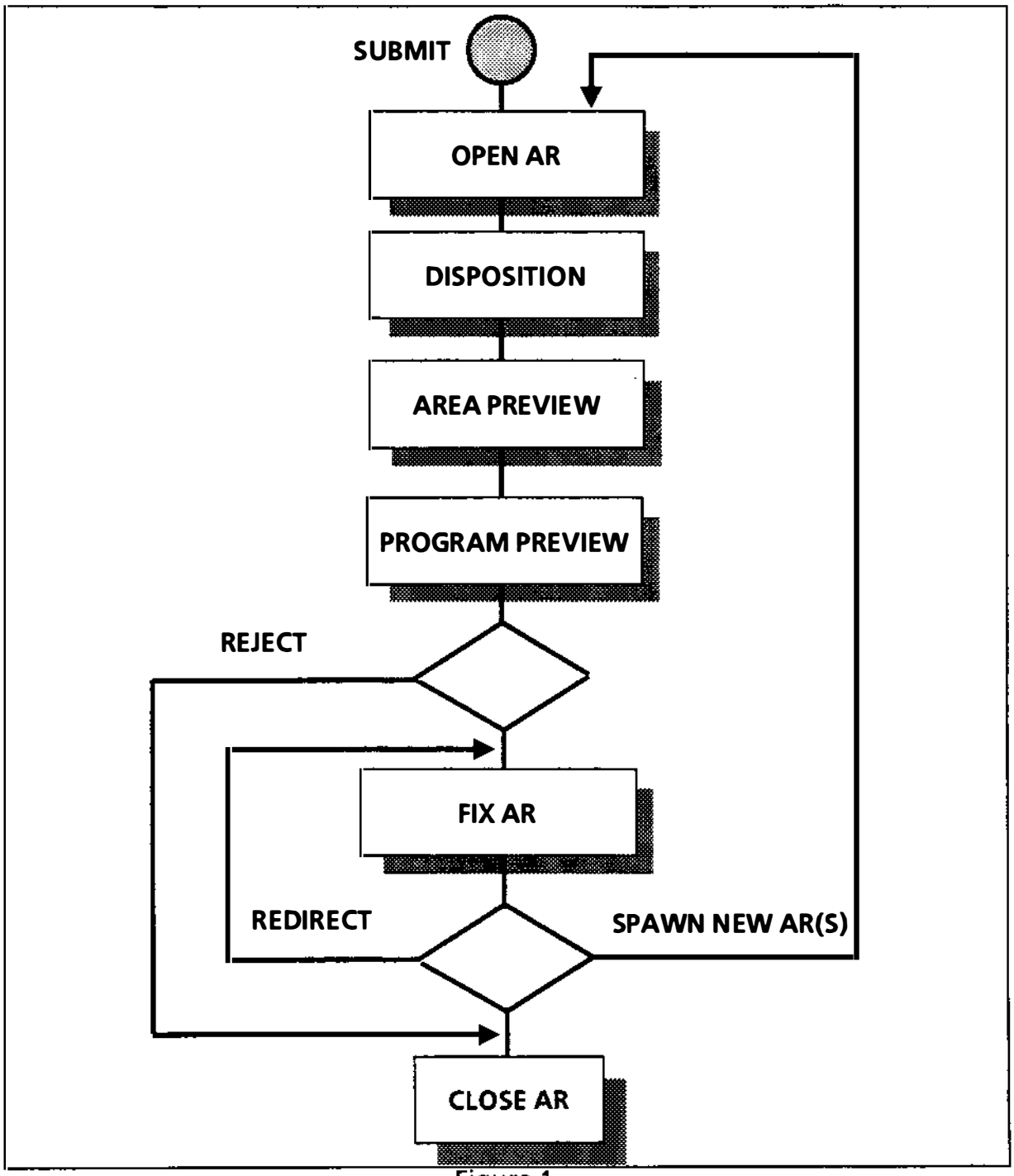



Figure 1.

The activities shown on Figure 1 are as follows:

- SUBMIT: This activity refers to the submittal of a defect report (AR). During the process definition phase it needs to be determined, who is allowed to submit ARs directly into this database. For example developers and testers would have direct access, while customers would not. Or, you might decide that you want to allow remote customer access via electronic mail.

- OPEN AR: This is the formal step when a database entry is created to track the defect report. During this phase the submitter will enter all known, relevant information, including submitter identification, detailed problem description, environment description, etc. (Core AR data).

- DISPOSITION: Just looking at the problem description, most of the time it is not obvious which part of the system was responsible for the failure. For example, even hardware errors can cause system crash, which problems supposed to be reported to a different organization, and they would be tracked in a different database. In the disposition phase an engineer will look at the report, and will make a preliminary assessment to determine which development group's responsibility will be to fix the AR.

- AREA PREVIEW: As a result of the disposition activity, a list of ARs are presented to the various development groups every day. They will review the AR in the context of the total work requirement for the group, and make a preliminary prioritization, based on technical preferences, difficulty, etc. The relevant facts are recorded in the database.

- PROGRAM REVIEW: The final prioritization is done by a special review board, which is more tuned to the requirements of the overall program. They will consider the area recommendations, but they have the power to override the area-recommended priorities, if needed. This board also has the power to reject an AR, in which case it will be automatically closed.

- FIX: The developer receives the AR, and start working on it. Further analysis might uncover, that the problem is not in his/her module. In this case the AR will be redirected, and it will show up in another area's task list. It is also possible, that the failure, reported in the AR is attributed to multiple problems, and as such it should be resolved in phases or by different people. In this case the developer will initiate ("Spawn") a new AR as well, and will note the relationship in the appropriate database fields. It is also the "Fixer's" responsibility to enter all, software metrics* related information into the database (For example time to fix, problem origination cause, problem origination phase, test phase when the problem was discovered, etc.).

- CLOSE AR: This phase marks the fact that the problem was fixed, and the AR became inactive. Of course it will be available for further statistical analysis, but it will not be shown on daily management reports.

---

* We recommend to start "small", i.e. define and collect the vital few metrics only. Later, when all the involved parties - support organization included - are comfortable with the system, you can gradually extend the range of collected metrics information.

Let's discuss the main roles of a defect tracking system in the software development process:

- First, it is an electronic means to facilitate interaction between a "submitter" and a "fixer". Again, these terms might mean different people in different organizations, and that is why so important to understand these roles and relationships in advance. In our case the organization did not interface directly with customers, and customer complaints were pre-screened and prioritized by another group before they were submitted to the development team for fixing. Similarily, at least two test organizations were testing the product, and they were both potential "submitters". Finally, developers themselves could submit ARs if they discovered problems or wanted to submit change recommendations during the course of their work. In the context of the "interaction" function described above, a problem or defect report has the following characteristics:

  - Represents an audit trail, a log of events
  - Loses importance after closure
  - The report can be free form text
  - "The more the better", i.e. the submitter should provide as much information as possible
  - Accuracy is important, but not critical, phone follow-up is allowed.

  To use defect tracking at least to this extent is a must for any software development organization, regardless of their SEI/CMM standing.

- The second role is to provide a metrics database. Collecting and analyzing metrics data is fundamental in "climbing the SEI/CMM ladder" and break out of the chaos of Level 1. The metrics function imposes somewhat orthogonal requirements on the defect tracking system, and again, these issues have to be fully understood before tool selection could be started:

  - Metrics collection targets current project tracking and future analysis
  - Enumerated or numerical entries are preferred over free text to enable statistical analysis
  - Query/Report Generation turnaround-time can be an issue
  - Accuracy is more critical.

Since the ultimate goal in our case is to build and maintain a database, we present the structure of an AR database record (Figure 2).

Verbal interpretation of the conceptual structure:

- An AR can come from only one submitter (But the same submitter can submit many ARs, of course...)

- During the life of the AR, many fixers might work on the problem (And fixers might work on multiple ARs as well...)

- The core data includes the basic information on the problem. This field might have its own structure, specific to the organization.

- The AR can be spawned by another AR, in this case it is considered a "child" AR, and there is an active pointer pointing to it from another AR.

- This AR can spawn other AR(s) and become a "parent" AR.

CONCEPTUAL RECORD-STRUCTURE OF AN AR DATABASE

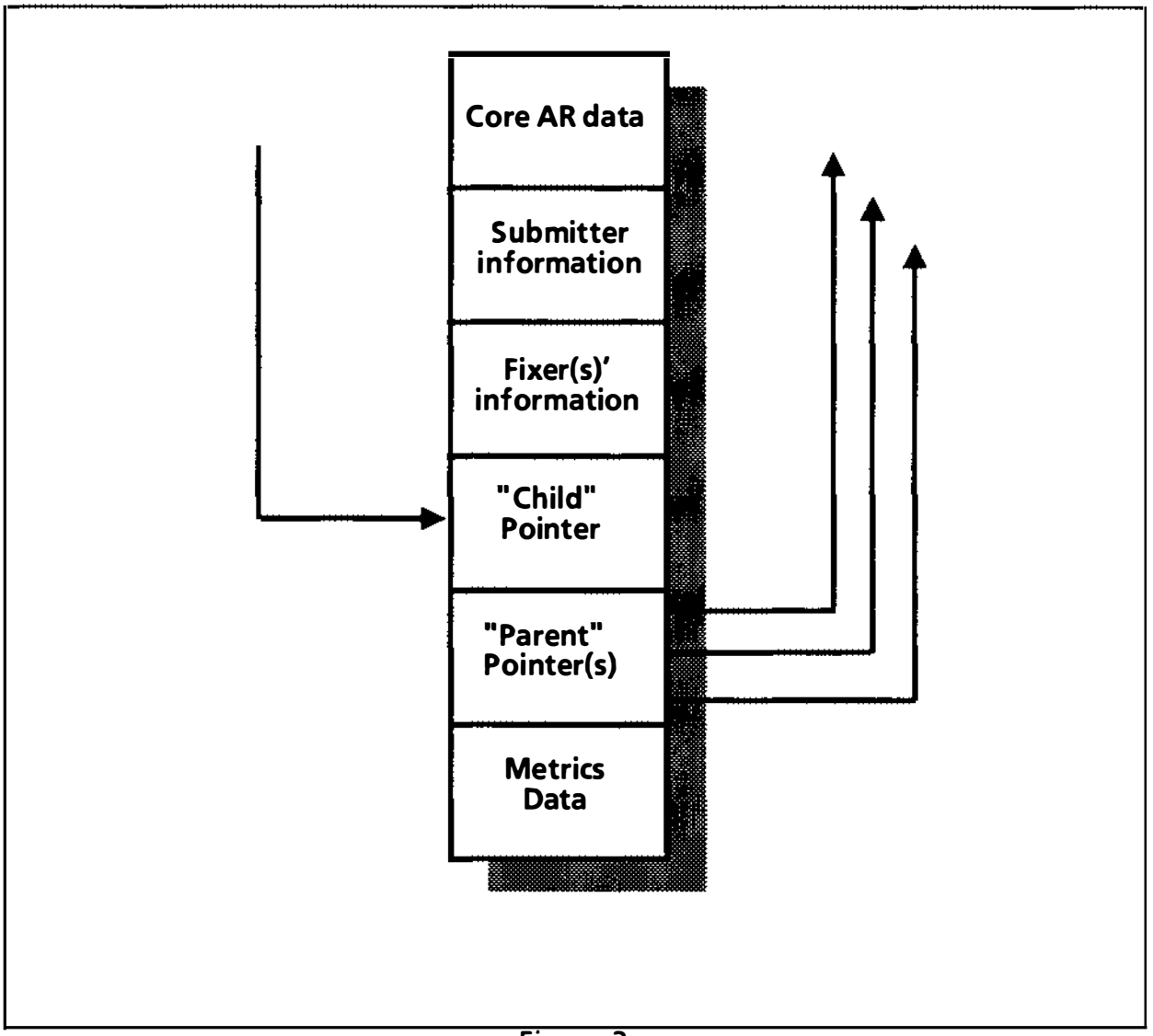



Figure 2.

Studying the defect tracking process, we concluded, we were looking for the following functions:

- o Log and store relevant data regarding defects, action and service requests
- o Examine, edit and query previously submitted defect forms and requests
- o Generate management reports
- o Guide and facilitate the AR tracking process.

In addition, we identified high level requirements as well:

- o The solution has to satisfy both tactical and strategic requirements:
  - Strategic: Has all the required features and will serve the organization for a long time
  - Tactical: Available immediately and facilitates legacy data transfer
- o The selected tool has to be cost effective
- o It has to require low maintenance effort
- o And finally has a high level of openness and customizability.

## 3. HOW TO DEFINE A DEFECT TRACKING PROCESS?

There are various recommended approaches and tools to systematically define and document a software process, for example **[3]** but their discussion is beyond the scope of this presentation. The less formal part - but nevertheless the most important one - is to collect customer requirements. The SEI/CMM approach recommends the creation of Process Improvement Teams, specializing in various KPAs, and they would have to find the proper channels and forums to identify customers and their requirements. We believe that the previously presented process can be used to "jump-start" the Process Improvement Team activites.

## 4. THE TWO-TIER PROCESS TO SELECT A DEFECT TRACKING TOOL

The process steps are as follows:

- Collect customer requirements, develop an all-inclusive list of features
- If appropriate: Evaluate "legacy" system against the current customer requirements
- If replacement decision was made, then decide on the strategy
- If the strategy is "buy", then collect information on available tools
- Determine the "vital few", critical requirements from the all-inclusive list
- Select a few number of "finalists", based on the available information
- Apply the all-inclusive list and execute an extensive benchmarking of the "finalists".

What triggers the whole process, i.e. why would organizations decide to replace a legacy system if they have one, or implement a defect tracking process, if they did not have it before? The second question is the easier one to answer. Studying the very extensive literature of software process improvement, or reviewing the experiences of highly successful companies, it is obvious that having a defect tracking process is a must. The size of the organization and the complexity of the products will determine if a manual process is adequate, or a sophisticated tool is needed.

The case of legacy systems is more difficult. Usually people might complain about various features (Or lack of ...), but they are comfortable with the tool, and might be resisting to change. For this reason it is even more important to identify and clearly communicate the shortcomings of the legacy system. For example, in our case we identified the following problems:

- While the legacy system had a - desirable - client/server architecture, both the server and the client required proprietary hardware and software
- Lack of proper data security and integrity mechanisms
- It was not a "true" dbms, it supported a simplistic view of the data only, and as a result, unnormalized schemas proliferated, i.e. data-fields were unnecessarily repeated
- The system became very slow when the size of the databases reached a certain limit
- The number of concurrent connections was very limited, users were often denied system access
- The reporting interface was very simplistic.

While we were able to develop workarounds for most of the problems, the first issue, the necessity for proprietary hardware and software became critical, because all external support was going to cease on these systems. This was the reason why senior management instructed our group to find a replacement tool.

---

[3] Defining Software Processes, SEI/Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, Copyright 1994 by Carnegie Mellon University

## 5. TOOL SELECTION STRATEGIES

o In-house development:

After going through a long analysis of the defect tracking process and collection of customer requirements, the temptation was great to develop a custom system. We decided against this option, and we are recommending the same to everybody else. Developing AR tracking software or database systems is not our mainstream business. Also, we were keenly aware of the fact that even if we would pull the development off, we would not be able to dedicate the necessary resources to support the software itself. Consequently it would not be prudent on our part to recommend the in-house solution, however interesting and exciting the project would be for the team members.

o Using software developed by another Xerox organization:

Various other organizations were flirting with the idea of developing a defect tracking system, and they were also very eager to find partners and customers for the project. One of the groups developed a very complex proposal for a problem management toolkit, and we were seriously studying this option. At the time of the review we felt that while it might become a nice system in the future, at that point it was only in a pilot phase, requirements were still debated and refined, and future support issues were not addressed at all. Last but not least, we also felt that at times when the corporation was streamlining operations and concentrated on core businesses and competencies, it would be an improper choice to pursue this option.

o Buying a tool from a 3rd-party vendor:

The last and most feasible option was to buy the tool. We were aware of the risks, i.e. companies can also go out of business, or promise a lot at the pre-sales phase and provide substandard service later. But we felt that if we did our homework, then the risks could be minimized. Also, we were not going to look for flashy or unproven technologies. For example some people raised objections, that the relational database management systems (rdbms) we targeted were becoming obsolete, and object databases represent the future trend. We felt we needed a solid, robust solution, and this was not the right time or environment to experiment with emerging technologies. Client/server computing and rdbms have reached the necessary level of maturity, while object databases have not.

## 6. THE INITIAL SCREENING PROCESS

We assembled a list of necessary, core features that the defect tracking tool must possess. Again, we used a different strategy than other organizations we surveyed earlier. The customary approach was to put together an all-inclusive checklist. As a result, the evaluation process became very laborious and often controversial. We chose the successive elimination approach. Any product which did not satisfy each and every requirement from the minimum list was eliminated from further consideration. Also, we did not schedule any actual experimentation with the systems at this time.

The team collected and processed information on ten products. Figure 3 illustrates the results. Presence of a black dot within the column indicates that the product does not adequately support that particular requirement. Any product containing one or more marks was eliminated from further consideration. There is nothing magic about the number 10, of course. Depending on the tool to be selected and the available benchmarking resources, one can proceed with either a shorter or a longer list, if appropriate.

**COMPARISON CHART**
**FOR INITIAL PRODUCT SCREENING**

| FEATURE DESCRIPTION | PRODUCTS | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **ARCHITECTURE** | | | | | | | | | | |
| Client/server architecture | | | | | | | | | | |
| Unix server platform, SunOS 5.x and above supported | | | | | | | | ● | | |
| Relational DBMS based, standard server | ● | | | | ● | ● | ● | | | |
| Open architecture, SQL compatible access | ● | | | | ● | | ● | | | |
| **USER INTERFACE** | | | | | | | | | | |
| Modern GUI, easy use (windows, buttons, menus, fonts) | | | | | | | ● | | | |
| Flexible customization of GUI | | | | | ● | | | | | ● |
| Client is good Open Look citizen | | | ● | | | | | | | |
| Progressive disclosure of fields, ability to nest dependencies | | | | | | | | | | |
| Automated completeness check during AR submission | | | | | | | | | | |
| **REPORTING** | | | | | | | | | | |
| Good reports (both standard and flexible ad hoc) | | | | | | | ● | | | |

Figure 3.

As we indicated earlier, this presentation is process oriented, and it is not an explicit or even implicit advertisement for any particular system(s), so we will not list the names of the vendors. Actually this is a very hot topic, and more and more vendors are entering this arena, so we really believe that only our process recommendations are convertible, and product recommendations would become almost instantly obsolete.

An interesting question was raised during the review of the paper: What to do if the initial selection - using the core features - leaves us with a null set of vendors? Since we did not run into this problem, we cannot talk from experience, but the parallel application of the following two approaches might work:

- Expand the circle of products, execute a thorough computer database search, and screen carefully current trade magazines for further products and product overviews. Consult with domain experts from various companies, like SEI or SPR for leads.

- At the same time review the checklist again, consult with your potential customers and make sure that the features listed in fact are necessary, and there is no opportunity for compromise.

The description of the minimum features is as follows:

- **Client/Server architecture:**
  C/S architecture offers several advantages. It provides an efficient division of tasks. The client can present the graphical user interface and handle user interactions. Meanwhile the database server is reserved for high-performance, large volume data processing. The C/S architecture also allows us to use different client platforms if needed, while retaining the same functionality.

- **Unix server platform:**
  This approach provided synergy with the mainstream hardware/software tools in the department, and allowed for a more optimal use of precious SysAdm resources.

- **Relational dbms based, standard server:**
  By using a standard dbms, the following features/functions would be automatically provided:

  - Data access optimization,
  - Data integrity control,
  - Transaction processing,
  - Concurrency control,
  - Security and authorization checking,
  - Backup and recovery.

  In addition we felt, that if we would have to migrate to yet another system in the future, having the information stored in a relational dbms would greatly ease the migration process.

- **Open architecture, SQL compatible access:**
  This capability would be essential in permitting connectivity with other software tools (Report generators, SCM systems, etc.)

- **Modern GUI:**
  A user-friendly graphical user interface is a must nowadays. The following window environments were important for us: X Windows (Using Open Look and Motif toolkits on the Sun/SPARC workstations) and Microsoft Windows for PC's.

- **Flexible customization of the GUI:**
  As we indicated earlier, the tool has to support the process, so the ability to customize the screens according to the process step's requirements is critical.

- **Client is a good Open Look or Windows citizen:**
  The tool should have the same look and feel as a typical Open Look or Windows tool, and should not interfere with the operation of any other window or tool on the desktop.

- **Progressive disclosure of fields, ability to nest dependencies:**
  This requirement refers to the ability of the data entry screens to change appearance depending on information which has already been supplied by the user during the earlier phases of the session; the change would instantly reflect the problem domain being described.

- **Automated "completeness check" during AR submission:**
  The tool should automatically perform as much syntax checking as possible, to ensure that all required information is entered in the appropriate format.

- **Good reports:**
  The ability for the system administrator and the users to create and run reports is essential. The tool should support some essential reports, but should also provide flexible, ad hoc reporting.

## 7. DETAILED EVALUATION:

We acquired and installed a copy of the three selected finalists for a detailed in-house evaluation. At this point we also created a detailed list of desirable features. Each feature item was also given a relative weight to reflect the importance of this attribute in the overall assessment. The weight values ranged from one to five, with one being relevant but least important ("Nice to have") and five being unconditionally important. Each feature was scored for the tools, where five represented the full compliance with the requirement. The final number was derived as a weighted sum of the scores.

The team spent about two months learning the tools. With occasional assistance from the vendors' sales team and hot-line facilites, all aspects of the systems were explored. Activities also included the design of sample schemas and porting old AR records to the new systems. During the entire process team members recorded their experiences on the evaluation sheet.

In the next phase we opened up the evaluation to the general developer community. This included activites such as demonstrations presented by the vendor's sales staff and also a one-week open house period, where our team members were available to give demos, address user concerns, exchange ideas and provide an opportunity for users to participate in live interactive sessions. All visitors were encouraged to fill out and submit the evaluation sheet, and their input was treated the same as the team members' scores. This was a key step to get user involvement and buy-in at an early stage of the project.

Figure 4 shows the summary results of the detailed evaluation, by feature category. The full table is included in the Appendix for your convenience.

**BENCHMARK RESULTS**
**SUMMARY**

| FEATURES | BENCHMARKED PRODUCTS | | |
|---|---|---|---|
| | A | B | C |
| UI Functionality | 92 | 65 | 77 |
| UI Customization | 43 | 35 | 25 |
| General Functionality | 105 | 101 | 97 |
| HW/OS Requirements | 82 | 88 | 84 |
| Documentation | 48 | 28 | 42 |
| Interoperability | 21 | 24 | 23 |
| Cost/Licensing | 40 | 30 | 25 |
| Report Generation | 36 | 14 | 41 |
| System Administration | 79 | 84 | 62 |
| Overall Impression | 20 | 15 | 15 |
| **PRODUCT TOTALS** | **566** | **484** | **491** |

Figure 4.

## 8. LICENSING ISSUES, COST COMPARISON

We knew cost was going to play a major role in the final decision, but we also knew in the final phase all vendors will try to bargain, so without going through an elaborate negotiation we could not gauge the real cost in advance. During the previously described detailed evaluation we used the figures the sales representatives provided in advance, without any negotiation.

**GENERAL PRODUCT ARCHITECTURE**

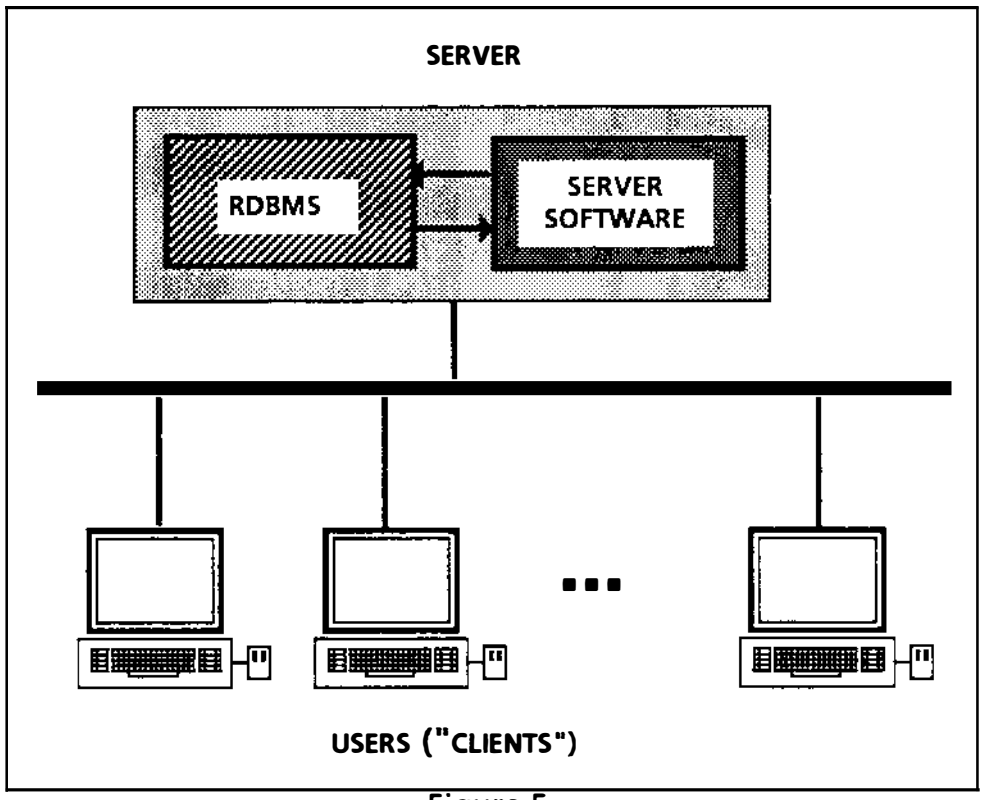



Figure 5.

Product architecture overview for cost assessment:

We have to assume, that the reader is familiar with the basic concepts of floating licensing, and also understands how users can be classified:

- **Number of actual users:** It reflects the number of installations of the client software on the net,
- **Concurrent users:** Users who are executing a database transaction at the same time
- **Potential Users:** It reflects the future growth of the development organization

All three products had the same general architecture from the cost and licensing's point of view (Figure 5). Usually querying the database without update is "free", and the products' licensing schemes do not limit the access of this kind. The licensing is tied to the number of concurrent users, and "concurrent use" really means concurrent update requests to the database. Concurrent access is measured on the database system itself. The handling of multiple clients and dbms access are processed by the vendor's server software, and not by the rdbms. As a result, all three finalists - unlike

some other, homegrown solutions we have also seen - required only one single rdbms license on the server. This architecture made all systems very cost effective compared to other solutions, where "n" concurrent client accesses required "n" instances or tokens of the rdbms user license. (Of course one does not have this issue, if willing to settle for a flat-file platform instead of a complete and independent rdbms server. Most of the products also can be acquired with such flat-file support as well...)

**USERS PROFILE**

| NUMBER OF USERS | | |
|---|---|---|
| **Potential Users** | **Actual Users** | **Concurrent Users** |
| 200+ | 100+ | 10-15 |

**COST FIGURES**

| INITIAL PER USER COST [$ K] | | | MAINTENANCE FEE FOR 10 CONCURRENT USERS [$ K/YEAR] | | |
|---|---|---|---|---|---|
| **Product A** | **Product B** | **Product C** | **Product A** | **Product B** | **Product C** |
| 2 (↓ ?) | ← 4.5 | ← 4 | 3 | 8 | 7 |

The relevance of the special symbols is as follows:

↓? As mentioned earlier, we did not initiate any negotiations with the vendors at this phase, but it was assumed that even with the one which offered the lowest cost there might be an opportunity to work out a special deal or a company-wide volume discount.

← These vendors were bidding very aggressively. In fact one of them was going to make a blanket offer, matching any other vendor's price, just to get their feet in the "Xerox door".

Despite the vendors' aggressiveness about the purchase price, the yearly maintenance fee was going to be a set percentage of the list price. The third table shows the disparity of yearly maintenance fees for the different products.

## 9. CONCLUDING REMARKS

We presented a tool selection case study, with as much generalization as possible. The final step - which we did not elaborate on - was to review the companies financial situation and do a careful reference check by talking to their current customers. Our team's recommendation was accepted by the organization's senior staff, and the tool was acquired and implemented. We believe that the key to our success and the ability to complete the benchmark in a timely fashion was the two-tiered approach, where first we defined the vital few criteria to narrow down the list of available tools, and executed a thorough benchmark of those tools only which passed pre-selection. We also believe that the demonstrated process and the attached checklists and tables will provide substantial help for others working on similar problems.

**APPENDIX**

| FEATURES | W 1-5 | BENCHMARKED PRODUCTS | | | | | |
|---|---|---|---|---|---|---|---|
| | | A | | B | | C | |
| | | Score | S x W | Score | S x W | Score | S x W |
| **UI Functionality** | | | | | | | |
| Easy to learn & use, not confusing to casual user | 5 | 4 | 20 | 2 | 10 | 2 | 10 |
| Modern GUI (Windows, Buttons, Menus, Fonts, etc. ) | 5 | 4 | 20 | 3 | 15 | 4 | 20 |
| Visually appealing | 4 | 4 | 16 | 3 | 12 | 4 | 16 |
| Ability to handle ad-hoc user queries | 4 | 3 | 12 | 1 | 4 | 4 | 16 |
| Allows alternate method of submitting ARs via EMail | 3 | 4 | 12 | 4 | 12 | 4 | 12 |
| Progressive disclosure of field choices | 3 | 4 | 12 | 4 | 12 | 1 | 3 |
| PRODUCT SUBTOTALS | | | **92** | | **65** | | **77** |
| **UI Customization** | | | | | | | |
| Entries allow enumerated items if feasible | 5 | 3 | 15 | 4 | 20 | 2 | 10 |
| Possible to further customize UI | 4 | 4 | 16 | 3 | 12 | 3 | 12 |
| Variety of UI templates available (Edit, Submit, etc.) | 3 | 4 | 12 | 1 | 3 | 1 | 3 |
| PRODUCT SUBTOTALS | | | **43** | | **35** | | **25** |
| **General Functionality** | | | | | | | |
| Client/Server architecture | 5 | 4 | 20 | 4 | 20 | 4 | 20 |
| Based on a standard rdbms server engine | 5 | 4 | 20 | 4 | 20 | 4 | 20 |
| Ability to nest multiple dependent choices | 4 | 3 | 12 | 2 | 8 | 1 | 4 |
| Has SQL compatible access or query language | 4 | 4 | 16 | 4 | 16 | 4 | 16 |
| Automated 'completeness' check before submission | 3 | 4 | 12 | 3 | 9 | 4 | 12 |
| Independent examination of data in the database | 3 | 3 | 9 | 4 | 12 | 3 | 9 |
| Security levels, selective access-rights supported | 2 | 4 | 8 | 4 | 8 | 4 | 8 |
| Notification when AR changes status | 2 | 4 | 8 | 4 | 8 | 4 | 8 |
| PRODUCT SUBTOTALS | | | **105** | | **101** | | **97** |
| **HW/OS Requirements** | | | | | | | |
| Solaris-2 version is available on the SPARC platform | 5 | 4 | 20 | 4 | 20 | 4 | 20 |
| Client/Server runs on the SPARC platform | 5 | 4 | 20 | 4 | 20 | 4 | 20 |
| Client uses Motif interface on the SPARC platform | 5 | 4 | 20 | 4 | 20 | 4 | 20 |
| Client runs on Intel platforms under Windows | 3 | 4 | 12 | 4 | 12 | 4 | 12 |
| Client runs on RS/6000 under AIX | 3 | 2 | 6 | 4 | 12 | 4 | 12 |
| Client runs on MAC | 1 | 4 | 4 | 4 | 4 | 0 | 0 |
| Server runs on Intel platform as well | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| PRODUCT SUBTOTALS | | | **82** | | **88** | | **84** |

| FEATURES (Cont.) | W 1-5 | BENCHMARKED PRODUCTS | | | | | |
|---|---|---|---|---|---|---|---|
| | | A | | B | | C | |
| | | Score | S x W | Score | S x W | Score | S x W |
| **Documentation** | | | | | | | |
| Well organized easy to understand documentation | 5 | 4 | 20 | 2 | 10 | 4 | 20 |
| Low-cost hardcopy documentation available for all | 4 | 4 | 16 | 4 | 16 | 4 | 16 |
| On-line Help function available | 2 | 4 | 8 | 1 | 2 | 3 | 6 |
| Customizable On-line Help function available | 1 | 4 | 4 | 0 | 0 | 0 | 0 |
| PRODUCT SUBTOTALS | | | **48** | | **28** | | **42** |
| **Interoperability** | | | | | | | |
| Data export to Spreadsheet, statistical,db, etc. | 4 | 4 | 16 | 4 | 16 | 4 | 16 |
| Integrated w/ other developer systems, i.e. SCM | 1 | 1 | 1 | 4 | 4 | 3 | 3 |
| Developer access of database through other tools | 1 | 4 | 4 | 4 | 4 | 4 | 4 |
| PRODUCT SUBTOTALS | | | **21** | | **24** | | **23** |
| **Cost/Licensing** | | | | | | | |
| Low acquisition and maintenance cost | 5 | 4 | 20 | 2 | 10 | 1 | 5 |
| Availability of Floating Licenses | 5 | 4 | 20 | 4 | 20 | 4 | 20 |
| PRODUCT SUBTOTALS | | | **40** | | **30** | | **25** |
| **Report Generation** | | | | | | | |
| Good reporting capability, standard & customized | 5 | 3 | 15 | 1 | 5 | 4 | 20 |
| Easy extraction of software metrics statistics | 3 | 3 | 9 | 2 | 6 | 3 | 9 |
| Users can generate their own reports | 3 | 4 | 12 | 1 | 3 | 4 | 12 |
| PRODUCT SUBTOTALS | | | **36** | | **14** | | **41** |
| **System Administration** | | | | | | | |
| Help Hotline is available | 5 | 4 | 20 | 4 | 20 | 4 | 20 |
| Quality of Pre-sales Support | 5 | 3 | 15 | 4 | 20 | 2 | 10 |
| Ease of Installation | 4 | 4 | 16 | 4 | 16 | 2 | 8 |
| Ease of conversion of existing database | 4 | 2 | 8 | 2 | 8 | 2 | 8 |
| Minimal Maintenance Effort | 4 | 3 | 12 | 3 | 12 | 2 | 8 |
| User installable Client Software | 2 | 4 | 8 | 4 | 8 | 4 | 8 |
| PRODUCT SUBTOTALS | | | **79** | | **84** | | **62** |
| **Overall Impression** | 5 | 4 | **20** | 3 | **15** | 3 | **15** |

# IMPROVING COMMUNICATION USING GROUPWARE

Brandon Masterson and Tim Fujita-Yuhas

## ABSTRACT

This paper describes a project's approach for improving software quality through better communication mechanisms (groupware). A groupware tool and its guidelines were created to develop software processes, communicate engineering methods, and ensure audit capability. Groupware tools can be used effectively to document processes, decision rationale and practitioner-level knowledge. This information archival allows continuous improvement and re-use of much of the development process and environment on other projects, thus raising the quality (consistency) of the resulting software. Groupware products fill a niche that e-mail and group meetings don't address.

On our project, we developed a groupware mechanism consisting of a tool called Forum and its usage guidelines. This relatively simple mechanism is used for gathering, archiving and communicating the project knowledge (process and methods). It reduced the amount of project-related e-mail exchanges and lowered the number of project meetings. It has gained a general level of acceptance of greater than 70% throughout the software engineering personnel. The mechanism attempts to improve software quality by defining and improving software processes, engineering methods communication while providing process audit capability. First, it supports the definition and rapid clarification of processes to attain better procedures and checklists throughout the project. Second, it provides better communication of engineering methods by having a central repository for technical solutions and rationale to be shared among current and legacy personnel. Finally the transparent revision control of accessible files ensures audit capability. The resulting mechanism which provides the three desired capabilities, provides a framework for process and tool re-use on subsequent projects.

Brandon Masterson
30726 NE 183rd
Duvall, WA 98019
bmasters@eskimo.com

Tim Fujita-Yuhas
10900 NE 8th St.
700 Plaza Center
Bellevue, WA 98004
timfy@saros.com

Keywords: Communication, Groupware, Lifecycle, Continuous Process Improvement, Feedback, Reusability, Process audit capability, Process Improvement, Productivity Improvement, Incremental Process Improvement, Technology Transfer

## BIOGRAPHIES

Brandon Masterson is a Senior Software Consultant with Cascade Engineering Services, Bellevue Washington. He has ten years experience with Software Engineering in the aerospace industry performing both the Software Design and Verification roles. He worked at Boeing on the B-2 Fuel Management System and then at Sundstrand Data Control on Flight Data Recorder and

Flight Safety programs before joining Crane-ELDEC. Areas of interest include: Structured Design, Process and Productivity Improvement and Re-use. Brandon graduated in 1985 from Oregon Institute of Technology, Klamath Falls, Oregon with a B.S. in Computer Systems Engineering Technology and an A.E. in Electronics Engineering Technology.

Tim Fujita-Yuhas is currently a Software Test Manager at Saros in Bellevue Washington. Previously, at Crane-ELDEC in Lynnwood Washington, he was a Software Process Engineer. He has eight years experience with real-time embedded software engineering. He has been a project lead as well as a member of a software engineering process group. Previously, he worked at Charles Stark Draper Labs leading software R&D projects as well as providing project consulting on software projects. In addition, he has taught courses on software engineering in academia and businesses. Tim graduated from Boston University with a M.S. in Software Systems Engineering in 1990.

## 1.0 INTRODUCTION

This paper describes a project's approach to improving Software quality by standardizing and improving software processes, engineering methods and process audit capability to provide repeatable, predictable results. These improvements were accomplished by using a groupware tool and guidelines which allowed rapid continuous process improvement on projects. The application of this mechanism on our projects, combined with the technical and process re-use, improved software quality while a significant staffing increase was made to the project.

## 2.0 BACKGROUND

With the growth in electronic documentation and communication, it is increasingly feasible to document a project's processes, the decision rationale for those processes, and practitioner-level knowledge necessary to apply them. Many projects continue to transfer this information orally via hallway conversations, team meetings, and group e-mail exchanges. These mechanisms become increasingly inefficient as the number of people on a project increases. In addition, the knowledge is not easily accessible to those who need it or is not updated frequently, the information archival was a wasted step and software quality is not improved. By automatically archiving the captured knowledge, inadvertent destruction or loss by practitioners editing is avoided. Thus, the capturing of this core knowledge along with the corresponding rationale, in a centrally accessible, structured and archived manner, allows continuous improvement and re-use of much of the development infrastructure (process and environment). This improvement and re-use on subsequent projects raises the adherence to software process and standards. Following well-defined processes consistently yielded consistent results.

### 2.1 INTRODUCE REPEATABLE AND PREDICTABLE PROCESSES

One of the first capabilities needed for improving software quality is the introduction of repeatable, predictable processes[1][2]. These processes often take the form of documented procedures and checklists (e.g. code review procedure, code review checklist, etc.). When applied, such processes produce two types of results: product-related quality comments and process-related quality comments. The product-related comments, typically referred to as defects, are the expected result of applying a standard review process to a product[3]. The process-related comments, however, often reveal ambiguities or problems in the applied process itself which can lead to varying levels of quality in the product[1][2]. These process-related comments or questions, need to be addressed to ensure the consistency of the applied process and thus the product quality. A communication mechanism is needed to allow the ambiguities to be documented and then addressed in a timely manner.

### 2.2 IMPROVE ENGINEERING METHODS VIA COMMUNICATION

A second significant capability is the improvement of engineering methods through better communication of practitioner-level technical information (e.g. standards) and rationale[4][5]. A software project is made up of many different people, separated by both distance and time. Improving the communication of evolving engineering methods among these people can

significantly improve the success of a project. Improvements in communication can be accomplished in the following areas:

- brainstorming
- consensus-seeking
- decision rationale documentation
- technical solution documentation
- on-line meetings
- training
- tool documentation
- on-line help
- continuous improvement feedback

Each of these areas improve engineering when applied appropriately, but often the resulting information is not documented and thus cannot be applied by the project during its sustaining phase or used by other projects. Better methods are needed to facilitate the gathering of both practitioner-level technical information as well as their rationale[6].

## 2.3 DEVELOP PROCESS AUDITING

The third capability in improving software quality is the development of a process audit capability. This is especially important for safety related systems where verification of applying a documented process is a concern. Processes must be defined, put under revision control, and made accessible to be used by practitioners. In addition, process documentation must be easily modified while still being under revision control to attain repeatable process results. The rationale for the definition and modification to processes should also be available for feedback to improvement activities. A communication mechanism is needed that provides both the desired accessibility and still provides the required revision control.

The first two capabilities focus on creating, organizing and then improving the project's information by providing controlled accessibility. The third capability satisfies the need for implicit revision control of the information being communicated providing the ability to audit the process. To improve software quality as defined above, all of these defined needs must be fulfilled in a single system which serves as both a communication and documentation tool.

## 3.0 PROJECT CONTEXT

To facilitate the communication which addresses these aspects of quality improvement, a project-wide system is needed for developing, archiving and communicating information. Further, it must do so in a user-friendly manner to avoid penalizing the information providers and users. On our project, we developed a continuous process improvement mechanism to meet this need. This "mechanism" consists of two parts: a tool called Forum and its usage guidelines.

Forum was originally developed to support rapidly evolving processes, methods and standards as well as solving day-to-day problems associated with a subsystem development program for the Boeing 777 airplane. It was developed in a prototype fashion as the needs and requirements for the tool expanded as usage increased. It was originally intended to assist in communication about

project issues and process definition. E-mail and frequent team meetings were not efficiently satisfying this need especially as we increased the number of project personnel from 16 to 45. There are other alternative commercial products that provide many similar features of this groupware tool, but few were available when this tool was created and the usefulness of such tools was not widely recognized[4][7].

There were significant challenges that characterized this development project starting with the requirement to staff the project quickly to achieve a compressed development and certification schedule (24 months). To further complicate matters, in-house personnel were unavailable, forcing a large contract personnel hiring, training, and teaming effort. This intensified the need to capture the core knowledge for the successful technical transfer and maintenance of the project.

The project was ultimately organized into four teams of software engineers, each team consisting of nine to fifteen people. Many of the processes and engineering methods used by each team were also needed by the other teams, thus efficiently communicating the development of this information was vital to avoid wasteful reinvention or conflicting processes and methods. Given this environment a mechanism that readily supported better group communication is essential. Forum and its guidelines were developed to address this situation.

## 4.0 ALTERNATIVE MECHANISMS

Our project found that use of Forum is a more effective means of communicating and documenting volatile information than combination of team meetings and e-mail. The tool was not compared with alternative groupware products such Lotus Notes since the complete tool requirements were not identified when the problem was first recognized. Essentially it was not fully known what was needed much like those who have never had e-mail do not know its full value until they work in an organization that uses it extensively. We needed to improve the communication and documentation methods. E-mail, meetings, organized directories, bulletin boards and threaded news groups were all considered, but failed to support the desired environment. Starting with a very simple feature set, Forum was developed in an iterative manner adding features as needed. Recent announcements of commercial groupware products such as Collabra Share and OpenMind with similar feature sets appear to be a recognition of need for productivity management tools[4][5]. This type of communication groupware will probably find a niche as a part of an e-mail or information management product in the future.

Forum provides up-to-date information in a complete, summarized living document format, which is created by logging the dialogue and debate for relatively important decisions as they are made. The key to documenting project dialogue and decision rationale to make it a side effect of the communication process, rather than an additional step. To accomplish this goal, the primary communication system for discussing, debating, and resolving problems on a project must also serve as a documentation and archival tool.

Forum usage differs from holding meetings. Group meetings require schedule coordination, the recording and distribution of minutes, and become inefficient with many people. It differs from e-mail, which is interrupt driven, deals only with the moment, and poorly supports group

discussions. E-mail archives are also usually incomplete since user's save different information, and issue summaries are not available without painstaking research. A communication mechanism such as Forum provides full discussion context and history while threaded news items, memos, or e-mail messages provide only a fragment of the dialogue.

In addition to capturing the information, the method of generating the information can also be completed more efficiently. Instead of holding meetings to discuss issues, the frequency and duration of meetings can be minimized since information can be drafted, reviewed and commented directly via Forum. This has the side benefit of documenting the rationale for resulting decisions. Communication tools, like Forum, are not meant to completely replace meetings or e-mail, but provides a more appropriate communication and documentation medium for larger group interactions.

## 5.0 TOOL OVERVIEW

The Forum tool and its usage guidelines combine to make a relatively simple mechanism for gathering, archiving and communicating the information of a project. It supports the generation, review and communication of new ideas and/or processes in a controlled manner. Specifically, Forum provides the following groupware capabilities:

- a mechanism for communicating dynamic information (project issues, corporate processes, etc.) needed by groups. It provides a centralized repository of information available for access.

- a single source of up-to-date information in a complete, summarized living document format that is easily accessible for viewing, printing or extraction.

- a "changed" indication to signal individual users that the information in individual files has changed. User's can keep up-to-date or they can wait until the issue information becomes relevant before viewing the new changes.

- a standard method for anyone to add or view information.

- offsite access via modems or network connections to support telecommuting or late night catching up.

- search capabilities for locating references at an outline or detailed level. This is essential for quick retrieval of answers to questions or lessons learned.

- a means of identifying a focal point for issues on a project, referred to as a Moderator.

## 6.0 TOOL FEATURE SET

The following paragraphs discuss the Forum features currently provided in detail along with a few examples.

- Change notification capability

  Since Forum allows everyone's participation as a contributor or reviewer by providing direct input access, information in the Forum files can change quickly. This is especially true for addressing new or controversial issues such as standards or process definition/interpretation. This dynamic property required Forum to provide a quick, easy means for a user to determine if changes had occurred since they last viewed the file. To indicate this, Forum adds an asterisk on a group's Forum menu (see example of a Forum file menu in Figure 1). Once the changes are viewed the indicator is removed.

VIEW REVIEW FORUMS

| | |
|---|---|
| CONF | Emulator Configuration Review |
| CR | *Code Review |
| DR | Design Review |
| IR | *Integration Review |
| RR | Requirements Review |
| TOOL | Tool Review |
| TR | Test Review |
| VCR | Verification of Change Requests |
| HELP | FORUM Help |
| INI | FORUM User Initials List |
| FO | FORUM CONTROL MENU |

Figure 1

- Search capabilities

  Forum provides several methods for searching the knowledge base. At a high level, it provides a command for searching the section titles of all the Forum files in a Forum group. If the desired keyword cannot be found in a high level search, then a detailed search command is also available which searches the entire text in each Forum file. This command lists the section titles along with any matching text to provide context.

  Forum also provides the capability to list the contributors or moderators within a Forum Group. To determine the contributors, a command is available that lists the initials and name of anyone who accesses a Forum Group. The initials come from a data file that Forum builds and updates as new users are introduced. To determine who the moderators are, a list command is provided indicating the currently assigned moderator of each Forum file.

- Editing and Browsing capabilities

In addition to querying the documented knowledge base of Forum files for specific information, the Forum tool also provides several commands for editing and browsing the Forum files. It provides a command for ASCII-only editing. Writers use a standard notation for comments that can be filtered to support report generation and viewing capabilities. An outline command is available which displays the Table of Contents from each file within a Forum group. Forum also has commands for viewing Forum files with or without comments included.

- Change monitoring capabilities

Forum provides a means for examining what has changed or how a Forum file has evolved over time. There is a command to view changes that is used most frequently in conjunction with the asterisk indicator to quickly view what changed in a Forum file to assess its importance. This capability is essential for prevention of information overload as the number of Forum groups and files increase. Without this, users would not use the tool as it would take too long to find and view the changes. Second, a similar comparison command provides a side-by-side differences view of the changes. With these commands, the user can specify the file version to compare the current file against.

Two other commands are provided to access previous file changes. A history summary command lists the version, change date, author of the changes and a added/deleted line count for every version of the Forum file. The other command allows the user to view the full file of the selected version.

- Report generation capabilities

Forum provides commands for the extraction and printing of information from Forum files for inclusion in reports. The commands can print change reports, print entire files with or without comments, and extract entire files with or without comments. For example, Customer progress reports were generated by printing entire files without comments such as Review Procedures and Checklists. In another case, technical coordination reports were generated by several members of the team and snapshots of the reports, again without comments, were sent to the customer as it was being developed.

- On-line help

At the Forum file level, there are two on-line help mechanisms for users. The first provides a brief command summary. The second shows a command summary along with advice on how to be a moderator, how to comment, etc. The table of contents from the on-line help is shown in Figure 2.

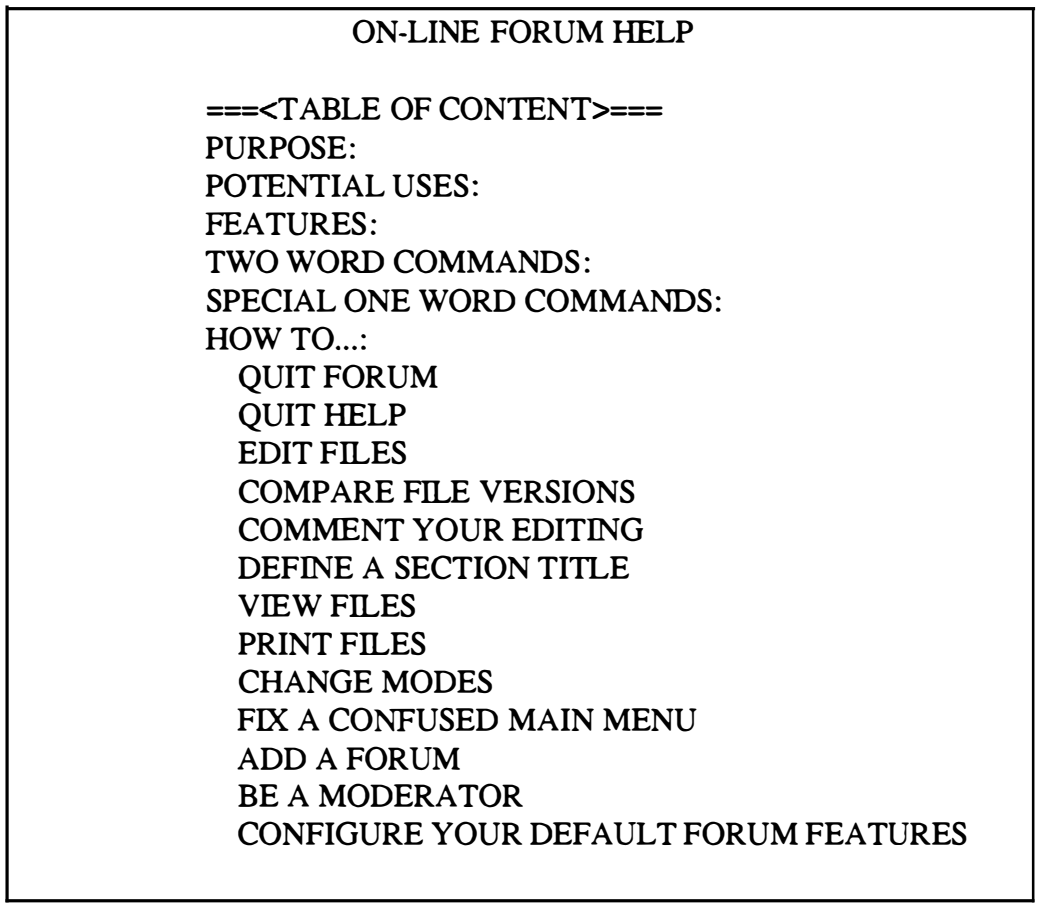



Figure 2

- User Customization

  Forum allows customization of the basic tools used to read or modify Forum files. Specifically users can define their favorite ASCII file editor, the file viewer, the print command options, and the file comparison tool.

## 7.0 TOOL USAGE GUIDELINES

In order for Forum to be effective it was determined that, several important guidelines must be followed. These guidelines include:

- each Forum file must have a moderator (e.g. focal point) assigned who is responsible for the information in the file. This person typically reviews questions or comments entered by other project personnel and responds by providing clarification or by refining the information in question. This person also summarizes other's comments once a direction has been selected on a given issue.

- each user must enter comments using a standard format which 1) allows the tool to filter them during extraction, 2) provides an quick indication of who the author of comment is and 3) clearly indicates this is a comment versus an information content addition to the file.

- Forum must be the main communication medium for project-related discussions between the software personnel. Forum cannot capture information or its modification if it is not used or if the majority of the personnel primarily use alternative communication methods such as e-mail or verbal exchanges. Since Forum is in a living document format it allows later users to join in its use with little loss of context or information while e-mail does not. This is analogous to why bulletin boards have Frequently Asked Questions (FAQ).

- All users need to be informed and encouraged to use the tool for the same purpose regarding a specific issue in a Forum file. Cross purpose clashes (having one person brainstorming for potential solutions versus while another is discussing solution evaluation) can be both frustrating and wasteful as our project discovered.

- The documentation and usage of practitioner-level lessons within a project must be encouraged. Typical problem solving should include checking Forum for previously documented problems and their solutions (e.g. Tips and Tricks). Similarly, it is desirable to use pre-existing methods and checklists as templates in the development of new, similar ones. This type of reuse can be a significant time saver on a project.

- A Forum administrator role must be assigned to regulate information categorization (organizing and restructuring files and groups) at all layers of Forum. As a project grows and progresses, information content in the Forum files will expand and be revised. This growth will require occasional recategorization of the information to avoid too many, or extremely large files. The actual recategorizations will usually be accomplished by the moderators, but an administrator is needed as an auditor. Lastly, the administrator must assist users who have problems with the tool.

On our project, these guidelines were generally followed for various reasons. First management emphasized and strongly encouraged the tool's usage. Second, the moderators who were often the main focal point for an issue kept the information accurate and formatted to reduce the number of questions directed at them (i.e. interruptions). Third, the Forum administrator would occasionally peruse the Forum files noting those that were significantly out of standard and then prompting the Moderators to clean them up. Finally, peers would occasionally ask users to clarify their inputs. Users which remained noncompliant were visited by management if needed. If this failed, the user was typically not considered for future responsible roles.

## 8.0 INFORMATION ORGANIZATIONAL STRUCTURE

Forum provides the following structured information hierarchy:

Projects Level
    Groups Level
        Files Level
            Section Level

At the Projects Level, the user selects a project from a list of projects (see example menu in Figure 3).

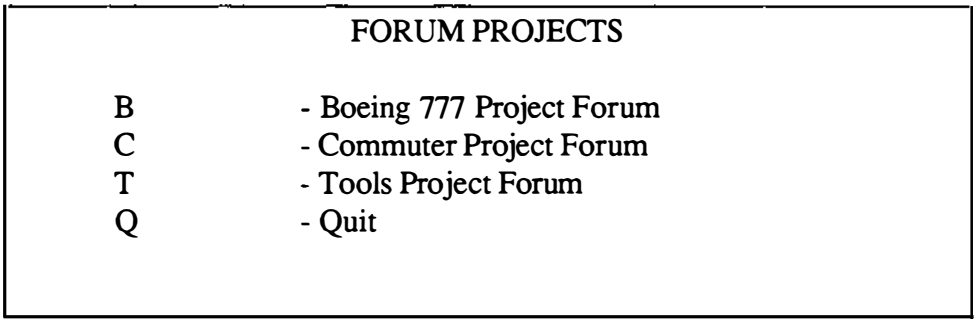



Figure 3

Once a user selects a project, Forum displays the next lower level, the group level. This level presents a list of Forum groupings within the selected project (see example menu in Figure 4).

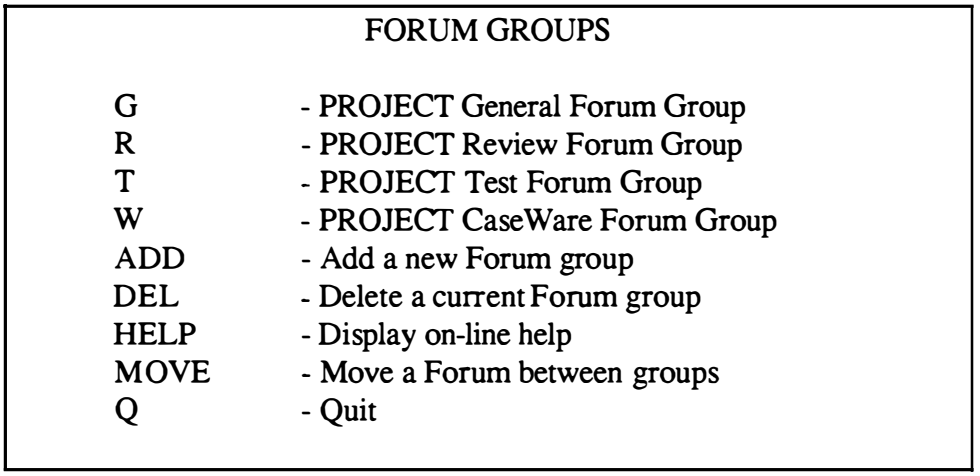



Figure 4

The level of information below a Forum group is the Forum files that are associated with the group. At this level, Forum files are stored in individual files that can be added, deleted or renamed. An example of the Forum files in a group is shown in Figure 5.

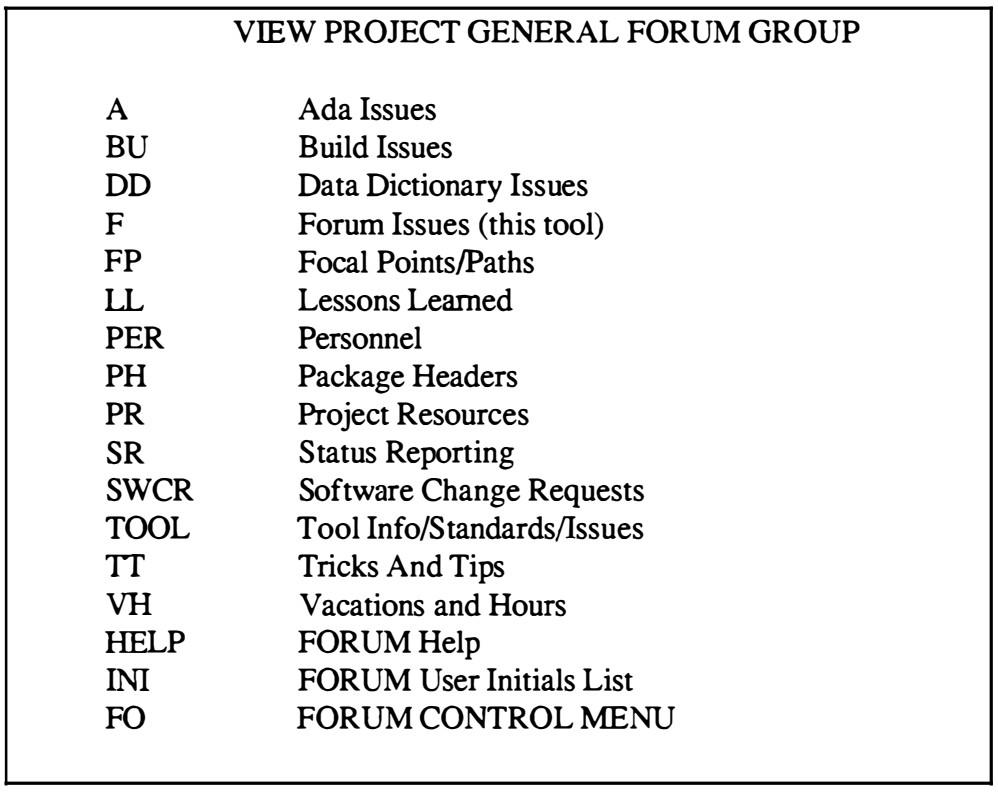



Figure 5

Within the Forum file are sections. The Section Level is simply the title under which information is documented. Refer to the Forum file excerpt in the Appendix for examples of sections.

## 9.0 USER ACCEPTANCE

The user acceptance of the Forum tool was typical of technology transfers of most new processes or tools. There were early adopters as well as laggards. On the project where Forum and its usage guidelines were developed, the mechanism received mixed reviews, especially during the early versions of tool and guidelines. As time passed, features were added and guidelines were clarified until most of the project personnel utilized the mechanism. It was accepted that 100% compliance was unrealistic, but their input if important could be inserted into Forum files by other users, such as the Moderator to maintain continuity.

Users who saw the most value in it included project leadership, software quality engineers (SQE) and software process engineers. Project leadership liked the mechanism since it provided visibility into work in progress and a place to organize and control a growing set of issues that needed resolution. SQEs appreciated the overall revision control on processes and standards, as well as the visibility into evolving processes. Some engineers recognized the ability to build a "core" knowledge base and liked being able to readily recover archived information.

Some users did not adapt to the mechanism for varying reasons. A few of the major ones included: not wanting to give up e-mail as main communication vehicle, dislike for the early

Forum user interface, and dislike for having to document and update the archived information (processes, standards, technical solutions).

The mechanism has transitioned to most new software product development projects with an estimated 80% acceptance level among the users. A large factor in the acceptance was the ability to peruse other project's core knowledge to look for processes and technical solutions that could be tailored for the new projects.

## 10.0 FORUM IMPROVEMENTS

Continued Forum development has not been extensive due to the end of the originating project and the addition of many of the earlier user requests. One recent change has been the addition of a Project-level menu which allows users to start at the very top of the knowledge base to search or scan for information (see example project menu in Figure 3). This improvement was a significant one as discussed in the USER ACCEPTANCE section. It encourages re-use of processes and technical information between projects.

One area Forum or a similar tool could improve on is the user interface layer. The addition of the addition of a HyperText Markup Language (HTML) would allow multimedia objects to be used in capturing core knowledge. A prototype was developed that allowed read only access. Adding this capability has several problems which must be addressed. The first problem is how to maintain revision control on hyperlinked files and multimedia portions of the forums. The last problem it how to provide a non-HTML interface that could be accessed from offsite using ASCII terminal emulation programs.

Another area Forum could be improved upon is in change notification by the addition of a notify list section to each Forum file. A user would be able to add their user name to a notify list section if they wanted to get notified via e-mail when a Forum file's content have changed. Users would be able to select from several levels of notification ranging from brief informational messages to having the entire Forum file sent to them. Moderators could use this feature to let them know they need to address a problem or comment. Users who depend on up-to-date technical information or who are just interested can be similarly notified. This feature would probably would win over the people who wanted to use e-mail as the primary communication vehicle.

## 11.0 CONCLUSION

This paper described experiences gained using a tool-assisted continuous process improvement mechanism. The mechanism attempted to improve software quality by defining and improving software processes, engineering methods communication while providing process audit capability. The mechanism supported the definition and rapid clarification of processes to attain better procedures and checklists throughout the project. It provided better communication of engineering methods by having a central repository for technical solutions and their decision rationale to be shared among current and sustaining engineering personnel. Finally the transparent revision control of accessible core knowledge ensured process audit capability. It was cost

effective to develop and use as evidenced by daily e-mail messages going from 80 messages a day to 20. Another indicator of success was that the number of team meetings and duration remained small even as the number of project personnel increased. The application of this mechanism (tool and its guidelines) improved software quality while a significant staffing increase was made to the project. In addition, it provided a framework for process and tool re-use on subsequent projects.

The ability to rapidly evolve and communicate processes and methods, building a core knowledge base as a by-product, would appear to be desirable in any software development situation. With the growing availability of groupware products, and the positive experiences on our projects, we conclude that utilizing some type of groupware mechanism would be beneficial for most software projects[4][7].

## 12.0 ACKNOWLEDGMENTS

We would like to thank Karen Parker, a Summit Design software engineer, for her initial input into the detailed features and potential use list as well her multiple reviews of this paper. We would also like to thank John Norris, a Crane Eldec software process lead, for his reviews as well.

## 13.0 REFERENCES

[1]: Paulk,M.C., Curtis, B., Chrissis, M.B., Weber, C. "Capability Maturity Model for Software (Version 1.1)", 1993

[2]: Humphrey, Watts S., Managing the Software Process, Addison-Wesley, Reading Massachusetts, 1990

[3]: Fagan, M., "Advances in Software Inspections", IEEE Transactions on Software Engineering, Vol SE-12, Number 7, July 1986

[4]: Baldazo, R., Diehl, S., "Workgroup Conferencing", BYTE, Vol 20, Number 3, March 1995

[5]: Yourdon, Edward, Decline & Fall of the American Programmer, Yourdon Press, Englewood Cliffs, New Jersey, 1993

[6]: Wallace, Scott, "Accelerating Engineering", BYTE, Vol 19, Number 7, July 1994

[7]: "Foundation for a New Class of Applications", http://www.lotus.com/notesdoc, Feb. 9 1995

## APPENDIX A

An excerpt from an example Forum file is shown below. In the example, the lowest level of information grouping, sections, is also shown (see Table of Contents).

TOOL INFO/STANDARDS/ISSUES
$Revision: 1.66 $

Moderator: Jane Engineer

Purpose: To provide a place to discuss tool needs, list available tools, store a template and define the tools standard.

[BUI - I thought I'd provide some guidance on the objective
of the project's efforts in generating this forum. The
tool standards and review checklist and process need to
be minimalistic in terms of effort to bring the
project's tools into compliance with the standard and
subsequently the effort to review such tools. The
quality of the tools must be sufficient to insure that
the tools can be maintained by others.]

===[TABLE OF CONTENTS]===
TOOL LIST:
TOOL NEEDS:
SUGGESTIONS:
BASIC GUIDELINES:
TEMPLATE:
CHECKLIST:
TOOL STORAGE:
==========

TOOL LIST:
See the Focal Points forum for the list of tools available.

TOOL NEEDS:

Source Viewer
Rejected by popular acclaim (sorry Jack).

SUGGESTIONS:

Standardizer tool

[YUB - The tool standardize_tool is now available for ksh
tools. It has some known limitations, but it
attempts to identify all the areas it over-pathed.]

BASIC GUIDELINES:

The following is the basic guidelines for In-house developed tools:

o Display a part number and version when invoked in the help mode. If deemed appropriate by the author, part number and version may be displayed on every invocation.

Part # - either official Company or Part number assigned by tools group.
Part number assigned by tools group would have the form: S23900-xxxxx

where S23900 is the department number.
xxxxx is a unique number for each tool.

Version # would be the form of: Vxx.yyn

where xx - is the major release identifier.
This is a 2 digit numeric field that starts at 01 and progresses up.

yy - a bug fix revision. For a major release this field shall be '00'.

n - an intermediate bug fix release for emergency fixes only. This is a one digit alpha character.

[YUB - This part number idea needs some support, like who is in charge of assigning these numbers? Where are they recorded, etc. ]
[IRU - Currently, there are no part numbers.]

**Paper Title:**

# Reducing Software Schedules with a Process-Based Configuration Management Model


## Author:

Karen Parker
Summit Design
9305 SW Gemini
Beaverton OR 97008
(email kparker@sd.com)



## Keywords:

- CaseWare CM Lifecycle Overhead Process Quality Schedule Software


## Biographical Sketch:

Karen Parker is a Senior Software CM/ Licensing Engineer currently employed at Summit Design in Portland, OR. This paper relates to experiences attained while employed at CRANE/ELDEC in Seattle, WA, where she was the lead of a tools and process organization. She has 11 years of experience in the software development field. Prior to coming to CRANE/ELDEC, she worked for IBM in the Federal Systems Division developing software for a satellite ground station. Her experience includes software requirements analysis, design, implementation, and test. Her accomplishments include writing and maintaining the software development plan for a large project (over 100 engineers) as well as designing and developing project databases for project planning and tracking. She received a B.S. in Mathematics from California Polytechnic University, San Luis Obispo in 1984.


## Abstract

Consider the question often asked in software development: Why can't ten engineers develop a product ten times faster than one engineer? Effort and schedule are known to have a non-linear relationship. What function is the additional effort serving if it is not getting the job done faster? This extra effort is generally referred to as overhead. The main source of this overhead is the complexity of developing many pieces concurrently and integrating them back together.

Assuming software schedules will not become less demanding, the success of a project depends on minimizing the overhead associated with adding more people.

A model for reducing this overhead must provide communication and tracking capabilities, configuration identification, and baselining mechanisms during the initial and rework phases of a project.

Such a model was defined and implemented by adding more effective project management, communication, and integration mechanisms to the CaseWare/CM (Configuration Management) 3.0 and CaseWare/PT (Problem Tracking) 3.0 products [1]. This development model was used and improved on several projects and contributed significantly to the success of the projects.

# Reducing Software Schedules with a Process-Based Configuration Management Model

Karen Parker
kparker@sd.com (503-643-9281)

## 1.0 Purpose

The purpose of this paper is to describe a model for software development that supports methods for productively managing the complexity of software developed by teams. One implementation of this model is discussed as an example to make the proposed model more understandable. This paper was written with the hope that the reader would consider the possible benefit of combining project management, communication and configuration management mechanisms to facilitate more efficient team development. Currently, these three areas are considered different domains and not often considered in a single solution.

This is not intended to promote a specific implementation or provide a specific solution. It is intended to point out the benefits of various mechanisms and share my experience with the software development community. It is intended for anyone who is developing or improving a software development process.

## 2.0 Introduction

The definition of the word "Complex" is, *consisting of a group of interrelated ideas, activities, etc. that form a single whole.* Any software implementation developed by more than one person is by definition complex. Even a simple application domain becomes complex when compressed schedules dictate that concurrent development and integration must take place. In *The Mythical Man-Month* [5], Brooks states: "Since software construction is inherently a system effort - an exercise in complex interrelationships - communication effort is great and it quickly dominates the decrease in individual task time brought about by partitioning." Since it is not always possible to keep the size of the team small, effective methods are needed to manage this complexity and reduce the overhead generated by these interrelationships.

This paper begins by examining the source of the overhead problem related to developing many pieces of software concurrently and integrating them back together. A solution is presented which provides a process model containing communication mechanisms and methods for tracking the development progress of the components through initial development and subsequent rework phases. Then several integration mechanisms are presented which reduce the overhead of identifying and baselining configurations. Finally, selected details of our specific implementation of the model are discussed. This model was iteratively developed over a two year period. It was applied over several projects, and then improved based on users' feedback. The implementation of the model integrates a database, a configuration management tool, and a task management system.

## 3.0 Overhead

There are three main sources of the overhead associated with adding more people to a development team. The first is the additional concurrent development which results in more communication needed to design and implement interfaces and keep a schedule. The second is the additional concurrent rework which is done after the initial development when problems are found. This additional concurrent rework results in a need for more communication between the developers, testers and managers to keep track of changes and outstanding problems. The third main source of over-

head is the additional developers involved in the integration task. This results in a quickly changing baseline and difficulty assuring that compatible inputs are provided by each developer at the correct time.

## 3.1 Concurrent Development Overhead

The overhead which is associated with concurrent development is primarily a result of the added communication and management needed in order to concurrently develop the parts.

The first major contributor to this overhead is the communication required among the team members to assure consistent interfaces. When an individual develops software modules, he designs both sides of the module interfaces. When interfaces are changed, he most likely will change both sides of the interface accordingly. But when the solution is distributed across a large number of developers, managing the interfaces becomes much more difficult. Communication is essential to develop correct interfaces and identify problems and solutions consistently. One way to reduce this overhead source is to make the communication a by-product of the development process. Tools which supply automatic documentation and notification of changes can greatly reduce the communication overhead.

A second significant contributor is the coordination of multiple resources simultaneously. As the number of engineers on a project increases, the complexity of scheduling and tracking progress increases. Unfortunately, the importance of this task also increases due to the potential for large numbers of tasks to be significantly behind schedule. By the time this is determined, it may be too late to shift resources to rectify the problem. One way to reduce this overhead is to provide better mechanisms for planning and accurately tracking the development progress so that estimates can be consistently updated and resources shifted appropriately. Tools which provide accurate, up to date status information can provide management with the capability to significantly reduce the overhead costs.

## 3.2 Rework Overhead

Another contributor to the overhead cost is the iterative nature of software development. After the initial implementation of software, there is often a long phase where problems are found which require the design and code to be reworked. This long rework phase makes the maturity of the software very difficult to determine since components which are considered to be completed often require rework. Although the initial development is often well planned and managed, the rework effort is usually unplanned and difficult to track. This makes it very difficult for management to apply resources correctly since a clear picture of the remaining work is not available. Inefficient application of resources, due to the lack of information, results in difficulty meeting schedules. Tools which provide accurate tracking of the rework tasks provide management with the capability to efficiently apply the resources to meet difficult schedules. In addition, the data can be used later to help plan the rework effort on subsequent projects.

## 3.3 Integration Overhead

Unlike the assembly of hardware, the integration of software is not a process which is done at the end of the development cycle. The software implementation must be repeatedly integrated throughout the development lifecycle to validate the components' interfaces.

During the first development phase, a requirements model is integrated and checked to ensure that all requirements have been identified and are consistent. During the preliminary design phase, design components are integrated to verify that the architecture is complete and consistent. Often during the detailed design phase, integration is performed when a compilable design language is used to validate the correctness of the interfaces and the module dependencies. Through the coding phase, modules are integrated into build configurations in order to validate larger pieces and avoid the overhead of stubs and drivers in testing. Each of these integration steps are repeated in the subsequent rework phase.

The integration task becomes more complicated when there are more developers involved for several reasons. First, there are more updates being made concurrently causing the development baseline to change very quickly. This

changing baseline makes it necessary to integrate more often. Second, there is greater difficulty in coordinating the integration to assure that each developer provides compatible updates at the proper time.

Because the integration process is performed repeatedly, the overhead of this process is a large contributor to the total project overhead. More efficient methods for performing the integration can significantly reduce the total overhead costs.

# 4.0 A Model for Reducing Overhead

The following sections describe a process model for software development that provides mechanisms for reducing the overhead costs associated with concurrent development and integration.

The model includes communication and tracking mechanisms during the initial and rework phases of a project that reduce the concurrent development overhead. It also provides the configuration identification and baselining mechanisms needed to reduce the overhead of repeatedly performing integration. It is based on the lifecycle of the work products (components) and configurations. See [2] for more information on "lifecycle modeling." The component's maturity in its lifecycle is based on review and early test criteria. The configuration's maturity in its lifecycle is based on the integration and development progress at a product level.

In addition to lifecycle, components and configurations also have other attributes associated with them. These attributes are used to indicate specific information used to manage the development process.

## 4.1 Managing the Component Development Process

### 4.1.1 Defining Component Maturity

The maturity of a component is an indication of how much effort is required to complete the development of the component. The *state* of the component is the component's maturity in its lifecycle. When a component's state is updated, it is referred to as a *state transition*. Figure 1, "Life Cycle of Components,' shows the valid state transitions for components. The following are the possible *state* values indicating the component's maturity level:

- working, checkpoint, visible, group, prototype -- indicates that the component is under development and could be in any state of consistency or correctness. A component is initially in this state when the development starts and returns to this state whenever a change is made. The purpose of multiple development states is to trigger notifications and provide a maturity classification for early development and integration. These states will be explained in more detail later in "Methods for Identifying Configurations" on page 9.
- review -- indicates that the component is going through a review process to verify its completeness and correctness. If defects are found, they are recorded as attributes of the component and the component is reworked in the *working* state. Once the review passes, the component is promoted to the next state.
- integrate -- indicates that the component is ready for formal integration and testing.
- released -- indicates that the component has passed existing verification criteria and is ready to be included in a release of the product.

### 4.1.2 Determining Development Progress

Development progress can be tracked using the lifecycle state and other attributes of the components. The transition of the components through the states represents its progress. Progress can be communicated via notification on state transitions and can also be determined by querying for the component's state. In order for the state of the component to be a true indication of maturity, there must be criteria for promotion. These criteria are referred to as *preconditions on the state transitions.*

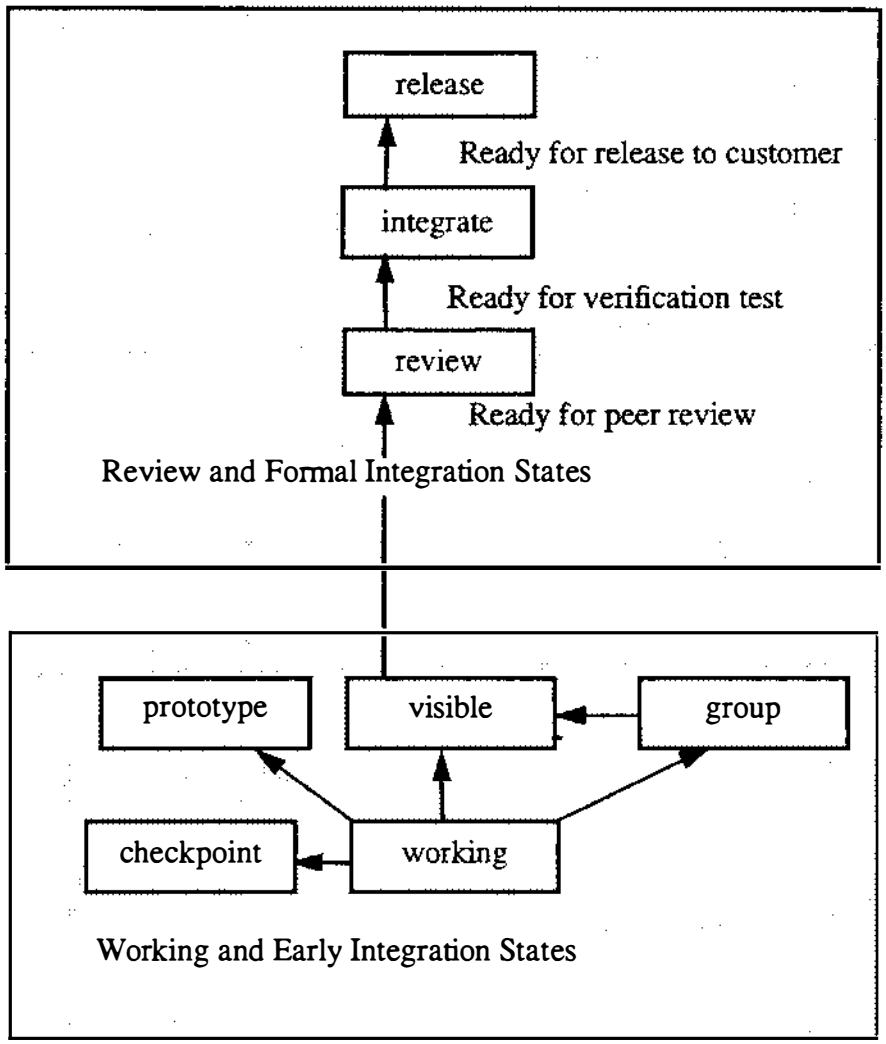



**FIGURE 1. Life Cycle of Components**

When the development begins, components are created and developed in the *working* state. As progress continues, the components are promoted to an informal integration state where they can be pulled into configurations and integrated to verify interfaces and compatibilities.

When the precondition for a review is met, a *reviewer* attribute is associated with the component to indicate who should perform the review. When the component is promoted to the *review* state, notification is sent to the reviewer indicating that the component is ready for review. This notification provides a communication mechanism which allows the review to take place in a timely manner.

As time passes, the manager can determine which components have reached the *review* state. *Summary rate charts* can be generated to show the number of components in each state as a function of time. When the reviews are performed, the review documentation is recorded as attributes of the component. Defects are recorded and rework is performed by creating a new version of the component in the *working* state. The manager can use the attributes to

determine the progress of the development effort and shift resources and project schedules accordingly. Figure 2, "Component Attributes,' shows some of the key attributes of a component.

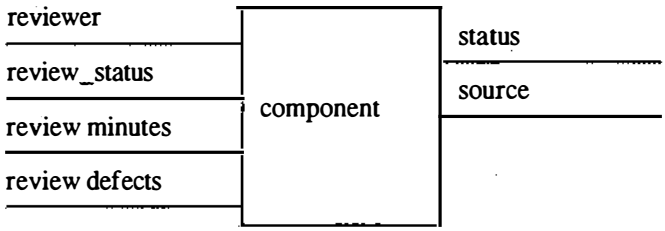



**FIGURE 2. Component Attributes**

Once the component has passed its design and code review, it is promoted to the *integrate* state. It is now available to be included in formal tests. At this point, the component is considered *complete* unless problems are found during verification or review of other components. Thus, all updates to the component are considered *rework*. Section "Tracking Rework" on page 6 describes how rework is controlled and tracked. Once a component is verified in a formal deliverable product, it is promoted to *released*.

### 4.1.3 Tracking Rework

Once all the components complete the initial development lifecycle, the project enters a different phase. In this phase, the formal verification testing process begins and errors are usually detected. Types of errors include incorrect requirements, unimplemented requirements, incorrectly implemented requirements, unspecified behavior, design errors, and coding defects. This phase is referred to as the rework phase. Typically, the rework phase is a large percentage of the project schedule and thus the capability to accurately plan and track this activity is critical. To reduce overhead effort in this phase of the project, the development model requires mechanisms for tracking the progress and estimating the rework effort.

Experience has shown that a good implementation for managing this rework effort is a task-based problem tracking system. In this system. problems or change requests are submitted for each proposed change. The set of tasks (i.e. requirements changes, design updates, code updates) needed to implement the change are associated with the change request. As a result, the rework effort can be managed on a task basis. Often in this phase, one task will require many components to be reworked. Thus reviewing the changes to a set of components is more effective than the individual component reviews done in the initial development phase.

Since many people may be involved in fixing a single problem, it is very important to have mechanisms in place to reduce the overhead resulting from communication problems. In order to facilitate this communication, the tasks themselves go through a life-cycle model. Figure 3, "Life-Cycle of a Task,' shows the optimum life-cycle for a task as well as the preconditions and notifications on state transitions.

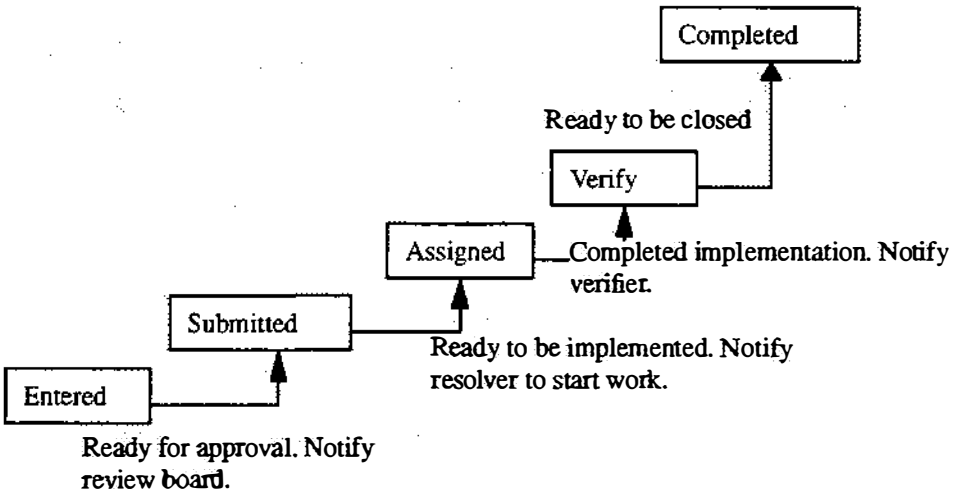



**FIGURE 3. Life-Cycle of a Task**

When a *problem* is discovered after the project has entered the rework phase, a *problem report* is submitted to a review board and notification is sent to the review board chairman. The review board decides if the problem should be deferred, rejected, or assigned. If the problem is not deferred or rejected, then a set of tasks that specify the proposed solution will be associated with the problem. This set of tasks will either be supplied by the problem submitter or an investigator assigned by the review board. If the solution includes the modification of components, those components are associated with the task. The tasks are then submitted to the review board for approval. Once the solution is approved, the review board assigns a *resolver, verifier, priority* and a *due date* for each task and the tasks are promoted to the *assigned* state.

Then the *resolver* assigned to each task has the authority to make updates to the associated components. When a task is assigned, the *resolver* is notified. When the implementation of the task is completed, the task's *verifier* is notified. If any problems are found, defects are documented and the *resolver* is notified that the task must be reworked. The communication mechanisms between the review board, the resolver, and the verifier keep the task moving and let everyone know exactly what needs to be done.

During this phase attributes of the task can be used to track the amount of estimated and actual effort, and duration of the change activity. Once the task is completed, attributes for the actual effort and completion dates can be filled in. Thus, reports can be generated that indicate the amount of rework effort estimated versus actual effort expended. This data can be used to determine what tasks are taking more time than expected and may improve estimates in the future.

The capability to reassign and re-schedule tasks is required to balance work load and increase team productivity. Since each task is associated with a set of components, reports can be generated which show the open tasks associated with a particular component.

The above model provides powerful mechanisms for planning, tracking, and managing the work-load. It also provides for post-project analysis of the tasks to determine what improvements can be made to reduce the amount of rework on future projects. If the data is controlled by a database, ad hoc queries as well as pre-defined reports are possible. This gives the managers the opportunity to collect and display the data in whatever way makes sense to them and provides a flexible management tool which works for many different styles of management.

## 4.2 Managing the Integration Process

Integration is an ongoing process throughout the development cycle. For a project to meet schedules, this process must be efficient. Time should not be wasted determining which versions of the components to integrate or investigating which versions were previously integrated. In order to efficiently manage the integration process, mechanisms must be provided that allow easy identification of the desired versions of components, assure communication between the development team, and provide the capability to baseline configurations.

### 4.2.1 Defining Configurations

When components are integrated, specific versions of the components are grouped together into configurations. A configuration is a set of components and processes that build a product. Examples of software products are a document, a binary executable, a test suite, and a support tool.

A configuration's maturity is determined by the maturity of the components included in the configuration. A configuration's stability is an indication of the improbability that configuration members will be changed. Like components, configurations also contain a state attribute. The following are the possible states for a configuration:

- `private`-- Indicates that the configuration can only be updated by the owner. A private configuration is used by a single user for the purpose of integrating components that are under development with other components that are more stable.
- `public` -- Indicates that the configuration can be updated by more than one developer. A public configuration is used by a team to integrate components that are ready for informal integration.
- `cfg_review`-- Indicates that the configuration is ready for a review and cannot be updated by anyone. A cfg_review configuration is used when the group of integrated components go through a review process together.
- `frozen` -- Indicates that the configuration cannot be updated. A frozen configuration is used as a baseline for subsequent testing and development.
- `released` -- Indicates that the configuration cannot be updated and contains components in the *released* state only. This state is used when a configuration becomes the formal *baseline*.

The *cfg_review* and *released* states indicate the maturity and stability of the configuration, while the *private*, *public*, and *frozen* states indicate only the stability of the configuration and does not specify the maturity of the configuration or its members. Figure 4, "Configuration State Transitions,' shows the state transitions for a configuration. Not all configurations pass through the *cfg_review* state because it doesn't always make sense for a configuration to be reviewed as a whole.

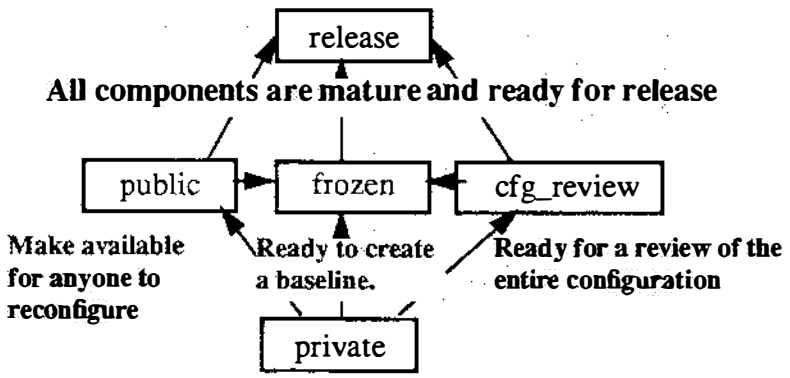



**FIGURE 4. Configuration State Transitions**

### 4.2.2 Methods for Identifying Configurations

Throughout the development lifecycle, many configurations are created. In order to identify the versions of the components that should be included in a configuration, an intelligent configuration construction mechanism needs to be provided [3]. When a user chooses to *reconfigure*, the configuration is updated with new versions of the components based on the configuration's selection criteria.

When a developer first starts development of his components he performs early testing of the component interfaces by including them in a *private* configuration. The purpose of a private configuration is to provide a mechanism for a developer to integrate the components which he is working on with selected versions of other components in the system. The configuration will consist of the developer's *working* components and interfacing components. The other components could be stubs and drivers or versions of the component under development. Since the components under development by other users become more mature over time, the user will need to reconfigure often in order to keep up to date. Therefore, a mechanism which allows the developer to quickly update his configuration with the latest versions, is necessary to reduce this overhead.

In order to make a version of a component available for early integration, the developers promote their components to a state that indicates that the component is ready. The early integration state transitions are shown in Figure 5, "Early Integration States" . That version of the component is not modified again so that the integrators can use it in a stable state. A *private* configuration is usually set up such that the latest version of components in a *visible* or higher state will be selected automatically on *reconfigure*. Components in a *visible* state have not been through a formal review process so the integrator has no guarantee that the component is mature.

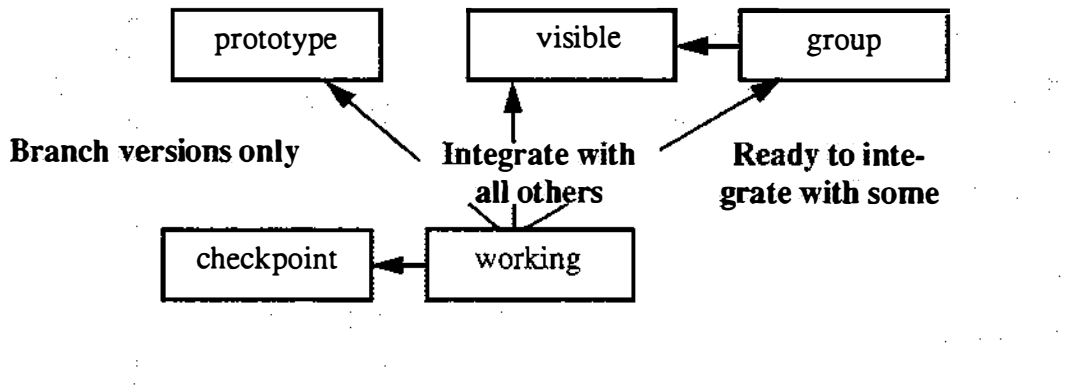



**FIGURE 5. Early Integration States**

Sometimes there is a need for some developers to integrate without making their working versions available to everyone. The *group* state is provided for this purpose. The selection criteria for a configuration must be intentionally updated to include components in the group state.

Once a component reaches the *visible* or *group* state, it is often helpful for the developers to know when a more mature version is available for integration. Thus a notification mechanism should be provided that allows a developer to subscribe to a particular component or group of components and receive notification when a new version becomes available.

The capability to informally integrate components being worked on by multiple developers is very powerful, but it can create more overhead if it is not done carefully. It is important that components are not integrated too early because errors in components can create problems in all the configurations that include the component. When a component is in the *working* state it is writable by the owner. It is not available for early integration by other developers.

This allows the developer time to work on the component to prepare it for integration. A *checkpoint* state is also provided that allows the developer to take a snapshot of the component by promoting it to a non-writable state, then deriving a new version to continue work. A component in the *checkpoint* state cannot be included in any configuration. Making a version of a component available for integration is a conscious decision and is done by the owner when the component is ready. Accepting new versions of components in a configuration is also an intentional step. It is done by the owner of a configuration when he is ready to re-integrate. This is important because it allows for early integration while still minimizing the effect of constantly changing components.

### 4.2.3 Methods for Creating Baseline Configurations

Baseline configurations are snapshots which keep a record of the exact versions of all the components which make up the configuration. Baselining is the process of creating and freezing a configuration.

The capability to quickly determine the difference between baselines and other configurations is important especially when problems are identified. Understanding the differences between configurations helps determine where errors may have been introduced. When a project does not have good mechanisms for baselining, it becomes very difficult to determine why the behavior of the product under development changes from one execution to the next and a great deal of time is wasted determining where errors have been introduced.

Initial baseline configurations are produced for the purpose of early integration testing. This often starts when the components themselves are mature enough to start the review process. Before reviews begin, it is beneficial to check interfaces and functionality of the components. The mechanism that provides for this type of integration is a *public* configuration. A *public* configuration cannot contain components in the *working* or *checkpoint* states. The components can be in any other state and therefore any phase of the component lifecycle. This common configuration is updated regularly with the latest versions of the components that are ready to be integrated. If a baseline is desired, the configuration can be promoted to the *frozen* state and a new configuration can be created for future integration.

Usually, creating baselines becomes important after the components have passed the review phase and are ready for formal verification. Official frozen configurations need to be provided on a periodic basis in order to begin the testing process. As new baselines become available, it is important to understand what has changed. This provides the tester with valuable information on where to concentrate his testing.

One important mechanism that is used to simplify the baselining process is release tagging. Once a configuration is frozen, the member components can be tagged to indicate the configuration that included them. Tagging allows developers who are working on changes to create a configuration that is identical to a specific baseline and apply only his changes to that baseline. A specific baseline can then be specified in the configuration rules to select only the version of components tagged for a particular baseline. The developer can then manually override the particular versions in order to integrate *working* components into the baseline. In addition, the configuration and the product that it creates can be tagged. This allows a mechanism for a test configuration to specify a particular baseline to test.

Once all the components are completed and the configuration is verified, the components and the configuration are promoted to the released state. A released configuration serves as a product archive. It is a snapshot of the final product.

The ability to baseline and systematically generate a new baseline of a configuration is important because it provides a mechanism to know what is being tested and ultimately what is delivered.

## 5.0 Experience Implementing the Model

An implementation of this model was done for a company in the electronics industry. A large percentage of this company's market is the aerospace industry. Since the airplane manufacturers are now requiring their vendors to provide integrated systems containing both hardware and software, software has become the critical path in meeting their product development schedules.

In order to meet aggressive schedules, it was determined that a large number of software engineers would be needed. The overhead resulting from the additional staffing needed to be minimized. Accordingly, a development process model was developed to provide better communication and tracking mechanisms during the initial and rework phases of the project to reduce the concurrent development overhead. It also provided the configuration identification and baselining mechanisms to reduce the overhead of repeatedly performing integration.

It was decided that the CaseWare/CM 3.0 (Configuration Management) and CaseWare/PT 3.0 (Problem Tracking) [2] products would be used as the foundation for the model implementation. These products provided a lifecycle-based development process, a task-based problem tracking system, and a strong configuration management platform to reduce the integration overhead. It was determined that the tools could be customized to provide the stronger project management and communication mechanisms required to implement the model.

Since the model source code is delivered with the CaseWare products, significant customizations were possible. The customizations included:

- Additional component states and state transition rules were added to assure that the state of a component is an accurate indication of development progress. In addition, some states were added to the configurations (or *assemblies*) for the purpose of adding flexibility to the integration and baselining process.
- A *reviewer* attribute was added to the components and the system required that a valid user be specified for each component before it could be promoted to the *review* state.
- E-mail notifications on state transitions were added to increase communication between the developers. For example:

    1. E-mail notification is sent to the *reviewer* when a component is ready for a review.

    2. Notification is sent to subscribers when a new version of a component is ready for early integration.

    3. The software lead can choose to be notified when reviews are complete.

- Additional attributes were added to implement the review process and the release tagging process.
- Extensive customizations of CaseWare/PT were done to provide a better task management system. The following are some of the customizations included:

    1. Added revised estimate for task completion date and effort which allows the resolver to revise the estimate without loosing the original estimate.

    2. Added estimates for task review effort and completion date to allow for the planning and tracking of verification work.

    3. Added a *task_deferred* state and "reactivate" date to allow work to be deferred. This allowed parts of a solution to be implemented while some parts were deferred.

    4. Added a *task_verify* state and review attributes to tasks to store the meeting minutes and findings of a review.

    5. Implemented auto-calculation of task review planned completion dates based on implementation planned dates and estimated review duration.

    6. Added a replanning capability allowing the lead role to reassign tasks and/or re-estimate completion dates and effort.

    7. Added task *views* for each task state and a task replanning view.

The total customizations represented over three person-years of effort. This paper does not describe the entire implementation, but touches on some of the key features of the model. Some of the other features of the implementation that were added to the basic CaseWare product include:

- Automated Unit Development Folders (UDF)
- Automated peer reviews (inspections)

- Test development environment to support an automated regression test capability
- Automated Version Description Documents (VDD)
- Integrated Alsys Ada build process

Although the CaseWare product has an implementation of a process model, the out of the box process model did not provide the strong project management and communication mechanisms described above. It also did not enforce a review process during the initial development or rework phase. Subsequent versions of CaseWare (now Continuus) provide a fuller feature set than the 3.0 version we customized [4].

The implementation of the model from scratch would have been too costly for this relatively small company. The model provided by CaseWare provided a starting point allowing a system containing parts of the model to be deployed relatively quickly. The ability to customize the model allowed for an environment to apply process improvement. When the process was not working well, the tool was often modified to provide either more automation of the existing process or a more ideal process. The use of the CaseWare product provided a solid CM platform and a comprehensive, process-based, integrated software development environment.

The deployment of this model and implementation on several projects was very successful. The largest software project ever undertaken at this company was completed on schedule. We attribute much of the success to the implementation of this model, which assured more consistency in the process and allowed much better visibility into the status of the project. Increasing the size of the team to accelerate the schedule was effective because management had the tools to apply the resources efficiently. The combination of strong communication mechanisms and integration tools allowed the team members to work productively together.

## 6.0 Conclusion

The success of medium to large size projects depends on minimizing the overhead associated with team development. The keys to reducing this overhead are better communication mechanisms and better management of the resources. Controlling the development with a process-based CM model provides a strong foundation for building these mechanisms.

This paper concentrated on two main strengths of the model. First, the model provides a mechanism for realistically tracking and communicating the status of the components at any time. This provides the manager the capability to apply the resources optimally to meet schedules. It also provides communication mechanisms that allow problems to be resolved concurrently without management intervention or coordination.

Second, it provides a flexible and productive integration environment which allows for a continual integration process throughout the lifecycle of the project. In order to utilize methods such as compilable design, incremental builds and early component testing, components must be repeatedly integrated throughout the development process. If the integration process is efficient, a great deal of overhead is avoided.

Our experience shows that a commercial configuration management and problem tracking system provides an excellent basis for building such a model. However, the customization of these systems was not trivial and fairly costly. The hope is that these commercial systems will provide more of these process and project management mechanisms in the future.

The project which inspired the development of this environment was very successful. It met all of its critical milestones and was delivered on schedule. The funding of the process and tool development was separate from the project funding since the process and tools were to be used on future projects. Accordingly, the return on that investment will not be seen until the environment is reused on subsequent projects.

## 7.0 References

[1]: CaseWare, Inc., *CaseWare User's Guide*, Part Number UG-031-012, 1993

[2]: Cagan, Martin and Wright, Alan, "Untangling Configuration Management, Mechanisms and Methodology in CM Systems" from proceedings of The 4th International Workshop on Software CM, May 1993.

[3]: Cagan, Martin and Wright, Allen, "Requirements for a Modern Configuration Management System", CaseWare Inc., February 1992.

[4[: Continuus Software Corporation, *Task Reference*, Part Number SDT-040-010, 1994.

[5]: Brooks, Jr., Fredrick P, *The Mythical Man-Month*, published (1975) by Addison-Wessley

## 8.0 Acknowledgments

I would like to thank Brandon Masterson and Tim Fujita-Yuhas for their multiple reviews of this paper.

*This is an index for all the articles in the Proceedings. Names of organizations and companies are in* CAPITALS; *names of computer programs and projects are in italics. Other entries are in* roman *type.*


## A

Abstract attributes, 132
Ada, 325, 345
ADP DEALER SERVICES, 254
Aircraft, commercial, 332
Alexander, Hilly, 254
Allred, Phil, 187
Alosi, Anthony C., 78
AMERICAN MANAGEMENT SYSTEMS, 21
Analysis errors, 358–74
Application as specification, 180
*AppTester*, 188
Assessments
  Diagram of current process, 412
  Limitations of, 48
  Process, 21–34, 45–56
  Repeating, 54
  Safety standard, 318
Audits, 421
  Project, 284–86
Automated testing, 178, 215, 281, 416
Award, Software Excellence, 397

## B

Backward reachable communication count, 345
Banking industry, 22. *See also* Commercial markets
Barcode scanners, 409
Barr, Valerie, 218
Baseline configurations, 474
Behavior abstractions, 130
Beizer, Boris, 148
Bell, Jeffrey M., 68
Beta testing, 277
Black-box testing, 129
BOEING, 176
Brainstorming, 412, 452
Brandt, Jonathan R., 409
Buddy system, 84
Build process, 282, 406
Business re-engineering, 398

## C

C++, 239
Capabilities, 74
Capability Maturity Model, 22, 45, 63, 259, 435
Card, David N., 115
CARNEGIE MELLON UNIVERSITY, 1
CASCADE ENGINEERING SERVICES, 449
*CaseWare*, 475
Cash bonus, 110
Causal analysis, 116–25, 286
Certification, 11, 127
Chakrabarti, Shankar L., 214
Change
  Leading, 59
  Promoting, 97
Change notification, 455
Checklists, 54, 451
  Defect types, 424
  Release, 286–87
  Review, 111
Chimpanzee, 184
*ClassBench*, 238
Cleanroom, 11–20, 126–147, 148–73
CLEANROOM SOFTWARE ENGINEERING, 126
Clear box, 130
Client/server systems, 347, 443
Commercial markets, 48
Commitment, 32
  Management, 287
Common problems, 305
Communication, 30, 400
  Distributed system, 343
  Dynamic information, 454
  E-mail, 475
  of Knowledge, 2
  of Metrics Data, 64
  Overhead of, 466
  and Project management, 66
  Technical, 451
  of Test results, 83
Competition, international, 26
Complexity, 342–357
  Managing, 466
Concurrent development, 467
Concurrent programs, 342–357
Conditional rule, 130
Confidence in product, 281
Confidentiality, 263
Configuration management, 25, 29, 403, 452, 465–77
  of Test data, 30
  of Test scripts, 184
Consensus, 103, 400
Consulting, 58
CONTINUUS, 476
Corrective activity, 359
Correctness, 129
Costs
  Correcting errors, 359
  Estimating, 97
  Process improvement, 255
  Support, 91
  Tools, 188–89, 445
Cover Certification, 154
Coverage, 218, 419
  Testing, 156
  Tools, 402
CRANE/ELDEC, 450, 465
Credibility, 83
Critical systems, 324. *See also* Medical software
*Cross System Product* (CSP), 364
Customer focus, 60, 108, 405

## D

Data abstraction, 130
Data design, 133
Data flow testing, 224
Database, relational, 443
Davis, Margie, 254
Debugger, lack of, 73
Debugging, 282
  Testing *vs.*, 155
Deck, Michael, 126
Decomposition, 5
Defect-free software, 69
Defect prevention, 127
Defect tracking system, 278, 434
Dent, Leslie A., 105
DEPARTMENT OF DEFENSE, 26
Developers, 106
*Deviate*, 181–85
DIGITAL EQUIPMENT CORPORATION, 423
Dirty dozen, 304–17

Disaster recovery, 35
Discipline, engineering, 1–10
Distributed systems, 342
Documentation, 385. *See also* Specification
  Design, 52, 280
  Lack of, 127
  Reading, 379
  Templates, 400
  Up-to-date, 454
Dynamic analysis, 225

## E

E-mail, 475
  Limitations of, 449
Ease-of-use, 311
Economic factors, 3, 5. *See also* Costs
*editres* protocol, 183
Effort, 74
Embedded systems, 409
Employee participation, 315
Engineering. *See also* Software Engineering
  History of, 1–10
  Science and, 7
Engle, Charles B., Jr., 11
Entropy, 361
Entry call diagram, 345
Environmental failures, 382
Epidemiology, 170
Error-based testing, 320
Estimation, 27, 52
  Costs, 97
  Errors, 359
Excellence Award, Software, 397
Execution path coverage, 227
Expectations, managing, 81
Experience, of developer, 85, 398
*EXPERT*, 228
Expert systems, 218–37
Expertise
  Delivery of, 59
  Nature of, 8
Extended propagation analysis, 320

## F

Facilitation, process, 102
Fact-goal path coverage, 227
Failures, random, 382
Fault-prone modules, 335
Fault-seeding, 320
Fault injection, 318–34
Fitness for use, 91
Florida Atlantic University, 335
Florida Institute of Technology, 11
Focus groups, 94
Food, 25
Food and Drug Administration, 376
Foreign countries, 21–34
Foreign languages, 189
Formal methods, 2, 73, 129
Forum, 449
Forward chain length, 347
Forward reachable communication count, 345
Frequency of failures, 426
Fujita-Yuhas, Tim, 449
Function points, 74
Functional programming, 72–73, 131
Functional testing, 179
Functional verification, 134
FURPS, 91

## G

Gefen, David, 358
Genetic algorithms, 182
George, Harry G., 176
Georgia State University, 358
Gifford, Jim, 187
Glossary, 156
  Testing terms, 112
Goal-Question-Metric, 60–67
Goal coverage, 227
Goal statement, 63
Graphical User Interfaces, 175, 177–86, 187–213, 214–17
Grass-roots team, 107
Greenleaf, Jenny, 397
Groupware, 449–64
Growth
  Organizational, 79, 106
Guidelines, 79. *See also* Standards

## H

Handbooks. *See* References
Hantos, Peter, 434
Hazard analysis, 384
Hazardous failures, 324
Health care, 376
Hewlett-Packard, 57, 89, 214
Historical metrics database, 75
History of engineering, 1–10
Hoffman, Daniel, 238
Hofstra University, 218
Hypermedia, 461

## I

Incremental development, 69, 128
Injury, 380
Innovation, 2
Inspections, 29, 73, 94
  Training, 403
Integration, 467
  Defects, 157
  Early, 70
  Testing, 154
Intermediate hypothesis coverage, 227
Internal software, 113
Intuition, limits of, 3
*Iridium*, 35

## J

Juran Institute, 290

## K

Kasik, David J., 176
Kelsey, Robert Bruce, 423
Key process areas, 27, 51, 259. *See also* Capability Maturity Model
Khoshgoftaar, Taghi M., 335
Knowledge
  Scientific, 2
  System, 83

## L

Language
  Foreign, 24–25
  Natural, 361
Leadership, 59, 307
Legacy systems, 126–147
Li, Raymond T., 434
Liability, 379
Libraries, 2, 239
Life cycle, 32, 79, 95, 256, 278, 385, 400, 410, 468
Linguistics, 361
Loyalty, 32, 88. *See also* Morale

## M

Maintenance, 81, 135, 410, 424

Management
Commitment of, 287
Middle, 255, 265
Senior, 49
Manual syntax checking, 160
Markov models, 142
Masterson, Brandon, 449
Mathews, Michael E., 342
Maturity
Component, 468
Engineering, 2
McKinney, Laura, 68
Mean time to failure (MTTF), 129
Measurability, 261
Measurement. *See* Metrics
Medical software, 375–388
Methodology, 109
Software, 287
Metrics, 69
Adopting, 57–67
Analysis errors, 358–74
Behavior-based, 319
Complexity, 342–357
Confidentiality of data, 263
Database for, 75, 438
Entropy, 361
Pace of adoption, 63, 401
Process, 262
Michael, C. C., 318
Milestones, 66, 80
Miller, K. W., 318
MIS, 358–74
ML, Standard, 72
Models and reality, 165
Modula-3, 182
Morale, 421, 88
Moreaux, Don, 89
*Motif*, 182
MOTOROLA, 35
MSR DEVELOPMENT, 276
Mutation testing, 320

## N

NASA, 325
Natural languages, 361
Nielsen, Jim, 35
Nondeterminism, 343
Notations
Conditional rule, 130
Software, 2
NOVELL APPLICATIONS GROUP, 187
Novice users, 177–86

## O

Olson, Timothy G., 290
*OpenVMS*, 423
Operational profiles. *See* Usage profiles
Opinion, value of, 102
Opportunity cost
of Error correction, 127
ORACLE, 402
Oracles, 143
OREGON GRADUATE INSTITUTE, 68
Organizational structure, 52
*OS/2*, 277
Overhead in development, 466

## P

PACIFIC SOFTWARE RESEARCH CENTER, 68
Parallel systems, 342
PARCPLACE SYSTEMS, 397
Parker, Karen, 465
*PARSE*, 423
Partition testing, 154
Path coverage testing, 224
Perturbation functions, 322
Pickard, Michael M., 276
Pilot projects, 110, 262
Planning
Process, 70, 109
Project, 27
Tests, 108, 416
Playback/recording tools, 178, 195, 215
Polk, Allyn, 174
Porting
Planning, 108
Tests, 189
Predictions, 69, 319
Completion, 75
Defect type, 424
Fault, 85, 335
Progress, 74
Prevention
Defect, 126
Priorities, 314, 424
Probabilistic testing. *See* Statistical testing
Process
Abstractions, 130
Defined, 52
Definition, 400, 421
Process *(continued)*
Development, 310
Fault-based, 319
as a Friend, 397
Historical, lack of, 398
Improvement of, 35, 46, 59, 79, 97, 105–14, 254–75, 409–21, 451
Modeling, 180
Planning, 70, 109
Repeatable, 451
Process design, 134
Productivity, 2
Program communication count, 343
Program conditional communication count, 343
Program transformations, 72
Project
Audits, 284–86
Management, 66
Notebook, 400
Planning, 27, 70, 109, 264
Tracking and oversight, 28, 52, 264, 313, 413, 470
Prolog, 228
Prototypes, 260, 398
Publications, plan for, 109

## Q

Quality
Design, 89–104, 129
Quality assurance, 28
Establishing, 277
QUALITY IMPROVEMENT CONSULTANTS, 290
Quality Improvement Core Team, 105–14
Quality Systems Review, 35

## R

Readiness for release, 427
Readiness of organization, 63
Real-time systems, 409
Reality and models, 165
Recording/playback tools, 178, 195, 215
Recursive datatypes, 72
References, standard, for software, 8
Regression testing, 143
Regulation, 383
Reliability, 2, 127, 157
Medical Software, 375–388

Reliability growth model, 142
RELIABLE SOFTWARE TECHNOLOGIES, 318
Repeatability, 260
Requirements, 403
  Changing, 110
  Customer, 309, 440
  Errors in, 358
  Gathering, 86
  Management of, 27, 52
  Specification of, 109
  Test, 80
Research organization, 68
Resource management, 3
Return on Investment, 110, 290–303
Reuse, 2, 238
Reviews, 29, 263, 316. *See also* Inspections
  Cleanroom, 134
  Design, 129
  Guidelines for, 111
  Post-project, 88, 399, 471
  Test scripts, 416
Revision control. *See* Configuration management
Rework overhead, 467
Risk, 171, 427
  Identification of, 81
  Managing, 82, 89–104
  Minimizing, 35
  Uncertainty *vs.*, 93
Robinson, Sally, 434
Robustness, 320
Roll-out, 114
RTCA DO-178B, 332
Rule-based coverage, 218–37

## S

Safety, 324, 452
SANGAMON STATE UNIVERSITY, 318
Saros, 450
Schedules, 52, 74, 312, 413, 465
Schurr, Steven C., 375
Science and engineering, 7
Scientific knowledge, 2
Scruggs, Darren, 187
Sensitivity analysis, 324
Shaw, Mary, 1
Shrink-wrap software, 188
Size of organization, 36, 45, 61, 79, 84, 106, 276
Small companies, 45, 106, 276
Smalltalk, 402
*Software Design for Reliability and Reuse*, 69
Software engineering, 1–10
SOFTWARE ENGINEERING INSTITUTE (SEI), 22, 45. *See also* Capability Maturity Model (CMM)
Software Engineering Process Group, 106, 255
Software Excellence Award, 397
Software Initiative, 57–67
Software management, 7
SOFTWARE PRODUCTIVITY CENTRE, 45
SOFTWARE PRODUCTIVITY SOLUTIONS, 115
Software science, 360
Specialization, 10
Specification, 11, 129, 401
  Application as, 180
  Changing, 279
  Executable, 217
  Functional, 109
  GUI, 214–17
  Implementation, 109
  Interposition, 139–40
  Language, 217
  Product, 108
  Recovery, 136–39
  Requirements, 109
SPECTRA-PHYSICS SCANNING SYSTEMS, 409
Spoilage, 427
Stability of releases, 427
Standards, 30
  Coding, 278
  Company-wide, 58
  Format for, 109
  Lack of, 79
  Processes, 451
  Quantifiable, 319
  Software Safety, 318
State box, 130
Statistical quality control, 11–20, 142–43
Statistical testing, 11–20, 142, 160–66
Status, reporting, 82
STEPHEN F. AUSTIN STATE UNIVERSITY, 276
Stimulus history, 133
Stratified fault seeding, 320
Stress testing, 142
Strigel, Wolfgang, 45
Subcontractors, 35–44, 311
SUMMIT DESIGN, 465
SUNSOFT, 174
Support
  Customer, 277
  Planning, 108
  Process, 265
  Product, 89–104
Supportability, 91
Synchronization, 343
*Synlib*, 216–17
SYNOPSYS, 105
Syntax checking, 160
System testing, 179

## T

Tape backup systems, 277
Task communication count, 343
Task conditional communication count, 343
Task scenarios. *See* Usage scenarios
Teams, 60
  Cohesive, 85
  Meetings, 453
  Overhead of enlarging, 466
  Quality improvement, 105–14
  Spirit, 421
Templates, 400
  Test plan, 112
  Test script, 421
Terminology, 156
Test data, 84
Testability, 324
Testing, 30, 78–88, 90
  Automated, 177–86, 215, 281, 416
  Beta, 277
  Component, 154
  Coverage, 156, 419
  Data flow, 224
  Debugging *vs.*, 155
  Early in life cycle, 279
  Efficiency, 420
  Error-based, 320
  Functional, 179
  GUI, 177–86, 188–213
  Improving process, 412
  Integration, 154
  Interface, 84
  Measuring quality, 128
  Mutation, 320
  Novice user scripts, 177–86
  Object oriented, 238–53

Testing *(continued)*
Partition, 154
Planning, 108, 416
Regression, 143
Repeatability, 163
Reportability, 420
Results evaluation, 184
Stress, 142
System, 179
Tools, 86, 178, 182–83, 187–213
Unit, 150, 154, 179, 281
Time zones, 25
Tools, 69, 315
Defect tracking system, 434
Extensibility of, 192
Importance of, 405
In-house development of, 192, 441
Remote access, 454
Selecting, 187–213
Testing, 86, 178, 182–83, 187–213
Trace table, 137
Training, 29, 261, 263, 315, 452
Translation, foreign language, 24
*TRUBAC*, 218
Tu, Shengru, 342

## U

UNDERWRITER'S LABORATORY, 331
Unit operations, 5
Unit testing, 150, 154, 179, 281
UNIVERSITY OF NEW ORLEANS, 342
UNIVERSITY OF VICTORIA, 238
UNIX, 443
Unpredictable users, 177
Usage profiles, 129, 161
Usage scenarios, 401
User errors, 380
User Interface Management System, 180

## V

Validation, 218–37, 385
VanSant, Jay, 397
Vendor relationships, 311
Verification, 227, 385, 470
VISA, 78
Visibility of problems, 425
*VisualWorks*, 402
Voas, J. M., 318

## W

Watson meetings, 286
Webber, Anton, 276
Whitten, Neal, 305
Wright, Gordon, 21
Writers, technical, 106

## X

*X-Windows*, 182
XEROX, 434

## Z

Zero-defect processes, 69, 159
Zimmer, Barbara, 57

## 1995 PROCEEDINGS ORDER FORM

## PACIFIC NORTHWEST QUALITY SOFTWARE CONFERENCE

To order a copy of the 1995 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 _____