FIFTHTEEN ANNUAL

PACIFIC NORTHWEST

SOFTWARE QUALITY

CONFERENCE

OCTOBER 28 - 29, 1997

Oregon Convention Center
Portland, Oregon

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

TABLE OF CONTENTS

Process Track — October 28

Technical Track — October 28

Management Track — October 29

Process Track — October 29

Technical Track — October 29

Preface

Members,
Hello and welcome to the Fifteenth Annual Pacific Northwest Software Quality Conference (a non-profit organization). Thanks for coming. By attending this conference you become a PNSQC member for one year. Here's my first report to you.

Our mission is to increase the awareness of the importance of software quality and to provide information and education opportunities to our members. I am delighted to say that we have assembled an excellent program again - thanks in large part to the diligent work of Sue Bartlett, Program Chair, to whom I am very grateful. I for one look forward to hearing many of the presentations.

As good as it is, we strive to improve and we need your help. Let me explain: Fifteen years ago, there was virtually no information available about software quality. Today, significant works are readily available. There are any number of software quality conferences, workshops, and consultants. Even though PNSQC is a non-profit organization, we choose to consider ourselves in competition with these other providers of software quality information. The alternative is to declare "mission accomplished" and to fold shop and we are not prepared to do that. Rather, we see PNSQC uniquely positioned to provide you, the industry, and the community with the best information about software quality. Tim Lister of the Atlantic Guild and co-author of Peopleware puts it this way:

> Every PNSQC is a gathering of true professionals: people eager to consider new approaches, to exchange ideas, and to question the status quo. It has been my experience, after three conferences, that there is no way to lecture at a PNSQC; the audience is too interested and informed to allow anything but a full forum discussion of the topic. If you are interested in the construction of high quality systems, meeting your peers at the PNSQC will be a stimulating and energizing experience.

But we have a problem. Recently the demand for new software quality information is outstripping the supply. In short, the number of paper submitted for PNSQC consideration has declined in recent years. As one of our key metrics, that is a trend that cannot continue. So here is the challenge that I put to you:

Challenge

As you participate in the conference today and tomorrow, be aware of what peeks your interest. Then select one software quality topic, research the literature, and build upon it. Write a paper and submit it for consideration at next year's conference.

Immanuel Kant said if he had seen farther than other men, it was because he stood on the shoulders of giants. With today's technology, we all have the opportunity to stand on the shoulders of giants - and there are giants among us. Take the time, write the paper. It is important.

Acknowledgments

Many people have contributed to the success of this year's PNSQC. Some volunteers have donated many hours of their precious time. Those people are recognized in the next section. Their steadfast belief in our mission, and their willingness to help, make PNSQC the premier software quality conference. Thanks also to Terri Moore of Pacific Agenda whose unflappability keeps us on track throughout the year as we plan and implement our events.

But it is the content not the process by which products are evaluated. Therefore, I would like to thank the presenters, invited speakers, and workshop providers for their contributions to the body of software quality knowledge.

Lastly, thank you, the members for choosing PNSQC. We, the PNSQC volunteers, hope that you enjoy the conference and take up the challenge. Contact any of the Committee members for more information.

Ian E. Savage, CSQE, PNSQC President and Conference Chair

PS If you would like to receive further PNSQC reports, fill out the Feedback Form and include your email address.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Ian Savage - President/Chair
Tiger Systems

Dick Hamlet - Vice President
Portland State University

Dennis Ganoe - Secretary
Wacom Technologies

Ray Lischner - Treasurer
Tempest Software

Judy Bamberger - Keynote Chair 1997
Process Solutions

Sue Bartlett - Program Chair
OrCAD, Inc.

Rick Clements - Keynote Chair 1998
Flir Systems

Laurie Duff - Software Excellence Award
ADP Dealer Services

Shauna Gonzales - Birds of a Feather
ImageBuilder Software

GW Hicks - Exhibits Chair
IMS, Inc.

Howard Mercier - Workshop Chair
Step Technology

Sudarshan Murthy - Publicity Chair
Tiger Systems

CONFERENCE PLANNING COMMITTEE

Chuck Adams
Tektronix

Hilly Alexander
ADP Dealer Services

Judy Bamberger
Process Solutions

Sue Bartlett
OrCAD Corp.

Kit Bradley
PSC, Inc.

Kingsum Chow
Intel Corp.

Rick Clements
Flir Systems

Dave Daurelle
North American Morpho Systems

Laurie Duff
ADP Dealer Services

Lynne Foster
Motorola

Dennis Ganoe
Wacom Technology

Shauna Gonzales
ImageBuilder Software

Dick Hamlet
Portland State University

Hewlett Packard

Warren Harrison
Portland State University

Karen Herrold
Intel Corp.

GW Hicks
IMS, Inc.

Dan Hoffman
University of Victoria, B.C., Canada

Craig Hondo
IMS, Inc.

Connie Ishida
Sequent Corp.

Mark Johnson
OrCAD Corp.

Bill Junk
University of Idaho

Randy King
Informix

Karen King
Sequent Corp.

Howard Mercier
Step Technology

Fred Mowle
Purdue University

Sudarshan Murthy
Tiger Systems

Aaron Putnam
OrCAD Corp.

Harry Robinson

Ian Savage
Tiger Systems

Eric Schnellman
Boeing

Keith Stobie

Bill Sundermeier
Flir Systems

Jim Teisher
Credence System

Craig Thomas
Sequent Corp.

Lee Thomas
Credence Systems

Miguel Ulloa
Intel

Scott Whitmire
Advance Systems Research

Barbara Zimmer
Hewlett Packard

PRESENTERS

Edward Addy
NASA/WVU Software Research Laboratory
100 University Dr
Fairmont, WV 26554
304/367-8353

Rick Anderson
Tektronix, Inc.
MS 38-248 PO Box 500
Beaverton, OR 97077-0001
503/627-2630

Shirley Becker
CSIS Dept.American University
4400 Mass Ave NW
Washington, DC 20016-8116
202-885-3275

James Bindas
Intel Corp
2111 NE 25th Ave MS JFT-101
Hillsboro, OR 97124
503-264-8869

Thomas Canter
Attachmate Corp.
3617 131st Ave SE
Bellevue, WA 98006
206-644-4010

Yuri Chernak
25 Zabriskie St Apt 3G
Hackensack, NJ 07601
212/506-6547

Shu (Billy) Chou
The Boeing Company
15752 SW 166th Pl
Renton, WA 98058
206/266-3301

Sadik Esmelioglu
Lucent Technologies
Room 2K-236 101 Crawfords Corner Rd
Holmdel, NJ 07733
908/949-3636

Gregory Hall
Univeristy of Idaho
Computer Science Dept
Moscow, ID 83844-1010
208-885-4077

Peter Hantos
Xerox Corp.
MS ESAE 375, 701 S. Aviation Blvd.
El Segundo, CA 90245
310-333-9038

Brian Janssen
ONYX Software
330 120th Ave NE
Bellevue, WA 98005
206/990-4080

Mark Johnson
Mentor Graphics
8005 SW Boeckman Rd
Wilsonville, OR 97070-7777
503/685-1321

Michael Kress
Boeing Commercial Aircraft
4316 229th Ave NE
Redmond, WA 98053
206/266-0545

Leslie Little
Aztek Engineering
2417 55th St #202
Boulder, CO 80301-2835
303/415-6232

Yves Mayadoux
EDF-DER
1, Avenue du General de Gaulle
Clamart, France 92141
33 1 47654275

Bill McCarty
Azusa Pacific University
Dept of CS, 901 East Alosta Ave
Azusa, CA 91702
818-815-5311

David Mundie
Texas Instruments
300 Oxford Dr
Monroeville, PA 15146
412-856-3600

David Mundie
Texas Instruments
300 Oxford Dr
Monroeville, PA 15146
412-856-3600

Harry Robinson
Hewlett Packard Company
1000 NE Circle Blvd.
Corvallis, OR 97330
541/715-7493

Robert Roggio
University of North Florida
Dept of CS, 4567 St John's Bluff Rd.
South Jacksonville, FL 32224
904-646-2985

Jarrett Rosenberg
Sun Microsystems
2550 Garcia Ave MS MPK 17-307
Mountain Park, CA 94043
415-786-5018

Wolfgang Strigel
Software Productivity Centre
#460-1122 Mainland St
Vancouver, BC V6B 5L1
604-662-8181

J Voas
RST Corporation
21515 Ridgetop Circle #250
Sterling , VA 20166
703/404-9293

Yingxu Wang
Research Centre for Systems
Engineering
176 Derby Rd
Southhampton, UK SO14 0DX
44 1703 319773

Karl Wiegers
Eastman Kodak Company
901 Elmgrove Rd
Rochester, NY 14653-5811
716/726-0979

George Yamamura
Boeing Defense and Space Group
P.O. Box 3999, MS 87-67
Seattle, WA 98124-2499
206/733-3762

Quality is Our Top Priority, Isn't It?

Mark Servello, *ChangeBridge, Inc.* & Ravi Apte, *Citicorp*

Abstract

Total Quality Management, Process Maturity, Malcolm Baldridge, ISO. With all of these quality-related programs, where is the quality? Senior managers initiate an "improvement program" then fail to staff it, fail to implement the recommendations, and wonder why they fail to see results. Why does this happen to technology organizations every day in every industry segment? "BECAUSE QUALITY IS NOT IMPORTANT IN TODAY'S BUSINESS ENVIRONMENT." Mr. Servello and Mr. Apte discuss what is important, and how to use that information to achieve quick and ongoing results using a "pay as you go" approach to building systems "better, faster, and cheaper."

Mark Servello *has 20 years of progressive experience in the development and operation of computer systems, ADO security, business information systems, scientific applications, and computer-assisted process control. Mr. Servello is an expert at Information Technology and business process reengineering and is a Master Instructor for the SEI's CMM training courses. He is a past director of the Society for Software Quality and past chair of the San Diego section of the IEEE. Mr. Servello is a founder and Vice President at Change-Bridge, Inc. which specializes in the effective delivery of technology to meet business objectives.*

Ravi Apte *is a Vice President and Division Executive with Citicorp/Citibank and is currently responsible for Technology Infrastructure for the Global Relationship Bank for Citicorp/Citibank. In this role, he manages and supervises all technology infrastructure activities for the Global Relationship Bank including Data Centers, Router Networks and continued deployment of a standard desktop. Under his stewardship the Singapore Regional Processing Center secured a ISO-9002 certification and the Asia Pacific Technology Group was assessed at Level 3 of SEI's Capability Maturity Model. Mr. Apte holds an Honors BSEE from the Indian Institute of Technology, Bombay (1970).*

Educating Software Engineers for Quality

Terry Rout, *Griffith University, Australia*

Abstract

There is a consensus emerging that the practice of software engineering has now attained a state of discipline consistent with the development of a true engineering pro-fession. At the same time, there is also a common view of the generic problems affecting software development, and of the actions and discipline required to address these issues. At the same time, however, we have been slow to develop the basic educational programmes needed to equip the new professionals with the competencies they require to meet the demands of industry.

In this paper, I have explored the current state of play, in an attempt to define the basic competencies required of practicing software engineers in today's industry. The source material for this comes from within the in-dustry itself, and from the professional bodies concerned with standards of practice. Based upon these competencies, I have attempted to define the "missing elements" from contemporary IT education. These include issues relating to peer review and teamwork; measurement; and the discipline of a defined process for development. The conclusion is that many of these factors, which have significant influence on quality, productivity and success in software development are missing or poorly addressed in much of our present educational effort. I have explored some of the solutions currently being developed in the educational area, and use these to point the way to the future.

***Terry Rout** is a Senior Lecturer in the School of Computing and Information Technology at Griffith University, Queensland, and is associated with the Software Quality Institute at the University. He has had extensive experience in the development and conduct of industry training courses in Information Technology over the past ten years. He is Chair of the Australian Committee for Software Engineering Standards, and has been a member of the Australian delegation to the International Committee on Software Engineering Standards from 1992 to 1997. He has been a member of the international management board for the SPICE (Software Process Improvement and Capability determination) project since its inception, working towards the development and validation of an international standard for software process assessment. He is the manager of the Southern Asia Pacific Technical Centre established to provide a regional focus for this work, and is the editor of two of the central components of the SPICE document suite.*

Terence P. Rout
Software Quality Institute
Griffith University, Queensland 4111
Australia

Phone: +61 7 3875 5046
Fax: +61 7 3875 5207
E-mail: T.Rout@cit.gu.edu.au
URL: http://www-sqi.cit.gu.edu.au/~terryr

PRESENTER: TERENCE P. ROUT

Title: Educating Software Engineers for Quality

Keywords: software engineering education/software quality/professional training

Abstract

There is a consensus emerging that the practice of software engineering has now attained a state of discipline consistent with the development of a true engineering profession. At the same time, there is also a common view of the generic problems affecting software development, and of the actions and discipline required to address these issues. At the same time, however, we have been slow to develop the basic educational programmes needed to equip the new professionals with the competencies they require to meet the demands of industry.

In this paper, I have explored the current state of play, in an attempt to define the basic competencies required of practicing software engineers in today's industry. The source material for this comes from within the industry itself, and from the professional bodies concerned with standards of practice. Based upon these competencies, I have attempted to define the "missing elements" from contemporary IT education. These include issues relating to peer review and teamwork; measurement; and the discipline of a defined process for development. The conclusion is that many of these factors, which have significant influence on quality, productivity and success in software development are missing or poorly addressed in much of our present educational effort. I have explored some of the solutions currently being developed in the educational area, and use these to point the way to the future.

PRESENTER: TERENCE P. ROUT

Title: Educating Software Engineers for Quality

Keywords: software engineering education/software quality/professional training

1. Introduction

> *"There are only two commodities that will count in the 1990s. One is oil and the other is software. And there are alternatives to oil."*
>
> *-Bruce Bond*

There are significant forces at work, both within the information technology industry and in the community at large, that are inevitably leading to the wider recognition of the professional status of software engineering. These factors include the increasing pervasiveness of software; the increasing recognition of its criticality in so many of its applications; and the increasing view that there needs to be dramatic improvements in the ability of the software industry to deliver products that work, on time, and for a reasonable price.

From within the industry, there have already been a number of initiatives in response to these pressures. They have largely been focused on building and extending organizational capability for software engineering; thus, we have seen the development and widespread implementation of improvement programmes based upon the Capability Maturity Model[1] and other models for software process assessment, culminating in the successful outcome of the SPICE Project[2] in developing an international standard in this field.

More recently, we have seen the emergence of models for developing professional discipline in individual software engineers. The Personal Software Process, developed by Watts Humphrey[3], is being employed to enhance the ability, or more correctly the professional competence, of software engineers in the way they approach the work of software. Results from the application of these techniques "indicates that a structured, disciplined, and measured personal software process can provide the guidance and feedback needed to help engineers improve their personal performance."

Despite these advances, the bulk of evidence indicates that our primary education for computing professionals - the software engineers of the future - has in most cases, not changed to reflect the new wave of practice. There are promising indications that a more intensive, engineering-oriented paradigm for software professionals is emerging; but the pace of change is too slow to keep pace with the rapid changes in industry demand. I want to ask, then, what must we provide the students in our universities today so that they can become effective in the industry jobs that most of them get after graduation? As a corollary, what should industry provide to help make this happen?

2. The Needs of Industry

As a first step in attempting to define the new paradigm for education of software engineers, I will examine the needs of industry, from a number of perspectives. There can be little doubt that such a new paradigm is needed. Humphrey observes "we have found that software engineers have difficulty adopting new methods. They first learned to develop software during their formal education and have since followed the same practices with a few adjustments and refinements. Since they are comfortable with these methods and have not seen compelling evidence that other methods work better, they are reluctant to try anything new." It would seem arguable, then, that a change in the approach to initial education of engineers might help to modify this culture and break the cycle.

The basic ideas of what can be described as "essential practice" for software engineering have been discussed for a considerable period. In 1983, Barry Boehm presented "Seven Basic Principles of Software Engineering"[4]:

Table 1: Basic Principles of Software Engineering
Manage using a phased life cycle plan
Perform continuous validation
Maintain disciplined product control
Use modern programming practices
Maintain clear accountability for results
Use better and fewer people
Maintain a commitment to improve the process

These carry the message that successful software engineering is fundamentally concerned with the development of discipline in product development - a recurrent theme throughout the emergence of the profession. Similar concepts are embedded in the models for improving organizational capability for software development: in the CMM, for example, the initial focus of improvement - the Repeatable level - concentrates on issues of project and product control. Again, Capers Jones reports that "the attributes most strongly associated with successful software projects included the usage of automated software cost estimating tools, automated software project management tools, effective quality control, and effective tracking of software development milestones." - further issues mainly concerned with discipline and control.

Table 2: The Ten Essentials of Software	
A product specification	A quality assurance plan
A detailed user interface prototype	Detailed activity lists
A realistic schedule	Software configuration management
Explicit priorities	Software architecture
Active risk management	An integration plan

Again, Steve McConnell[5] identifies "Software's Ten Essentials", shown in Table 2 - more of a product focus on this occasion, but still the emphasis on discipline and integrity of product and process.

A summation of all of these various approaches is probably best found in recent work to define the "Fundamental Principles of Software Engineering", undertaken at the International Software Engineering Standards Symposia in 1996 and 1997, and led by Pierre Borque and Robert Dupuis[6]. A set of candidate principles was established using a Delphi approach, and polled to identify those for which consensus could be reached. Following this poll, general consensus was reached that seven of the candidates met the criteria for recognition as fundamental principles.

Here, we find less emphasis on the "management" issues of discipline and integrity that characterize the other approaches. These fundamental principles seem to provide a firmer foundation for identifying the competencies required of software engineers.

Table 3: Fundamental Principles of Software Engineering
Since change is inherent to software, plan for it and manage it.
Invest in the understanding of the problem.
Since tradeoffs are inherent to software engineering, make them explicit and document them.
Uncertainty is unavoidable in software engineering; identify and manage it.
Set quality objectives for each deliverable product.
Establish a software process that provides flexibility.
Minimize software components interaction.

From these various sources, we can attempt to derive a small set of basic competencies that software engineers should have. These include:

A detailed understanding of change and configuration management as they affect software development

An understanding of the tools and techniques for specifying and achieving quality objectives for software products

An approach to software construction that integrates design and coding.

A practical understanding of the tools and techniques for management of software development projects.

Practical experience with techniques for verification of software throughout the life cycle.

An understanding of the principles and importance of reuse in software development

An understanding of the principles and application of measurement of software products and processes

It is my contention that most, if not all, of these basic competencies are poorly addressed in the bulk of our educational programmes today; this must change if we are to address the needs of industry in the future.

3. IT Education - the View of the Profession

In attempting to evaluate how well the Universities are achieving these basic competencies for the new profession, there are two routes we can follow. In the first place, the professional societies - the ACM, IEEE, BCS, ACS and the like - all have undertaken work to define core curricula for accreditation of courses meeting acceptable professional standards. In Australia and the UK, at least, these curricula form the basis for formal course accreditation as meeting basic criteria for membership of the societies. It is of value to examine these curricula to see how they match up against the demands of the new profession.

There are a number of caveats here; mainly they relate to the fact that most of the current curricula are under active revision, and that the issue of growing professionalism in software engineering is a key factor in these revision activities. It has recently been announced, for instance, that "the ABET has asked the IEEE to provide an analysis of the feasibility of accrediting software engineering programs in the United States".[7]

The ACM and IEEE have established a Joint Steering Committee for establishing software engineering as a profession. As a key activity for this initiative, a pilot survey was conducted of a range of competencies and knowledge areas, to identify views on the level of understanding needed for different levels of seniority in the profession. The knowledge areas included in the pilot survey were:

Algorithm Complexity	Caches	Client Server
Computer Peripherals	Data Management	Data Models for Databases
Database Administration	Database Performance and Capacity Planning	Database System Fundamentals
Database Systems	Data Structures	Device Drivers
Distributed Systems	Effort Estimation	Kernels
Power Management	Project Management and Planning	Real Time Systems
Risk Management	Software Quality Assurance	Static/Dynamic Linking
Transaction Properties		

The results from this survey are preliminary and tentative only; the committee does put forward, as "possible recommendations", however, the following:

> *Education for novice software engineers probably should not include high-level evaluation or management skills. Words like "prove" and "evaluate" are much less likely to be the verbs used in tasks for Novices than in tasks for Experts and Specialists.*
>
> *Current curricula for software engineers could be reviewed in the light of survey results -- are too many abstract concepts and high-level skills being taught to entry-level software*

engineers? If so, it is likely that those skills and knowledge will atrophy. Training research indicates that the opportunity to practice trained skills on the job is essential to the retention of those skills.[8]

It is unfortunate, however, that many issues identified in comments such as those we have examined earlier are not included in the areas of knowledge covered by the survey. For example, while the survey found limited need for understanding of estimating by novice software engineers, the question of familiarity with concepts of measurement - on which estimation depends - was not included in the survey. The eventual outcome of the Joint Committee will be - among other things - a Common Body of Knowledge for Software Engineering; it is likely however, that this is still some time off.

In Australia, a draft "Body of Knowledge for Information Technology Professionals" [9] has been developed by the Australia Computer Society, and is currently released for comment. This would seem to be the most advanced and recent of the curriculum efforts by the professional bodies, and it is worth examining it in more detail.

The ACS document is intended to cover the full spectrum of occupations in the domain of information technology; software engineering is one limited portion of this scope. Figure 1 shows the intended scope of the ACS BOK.

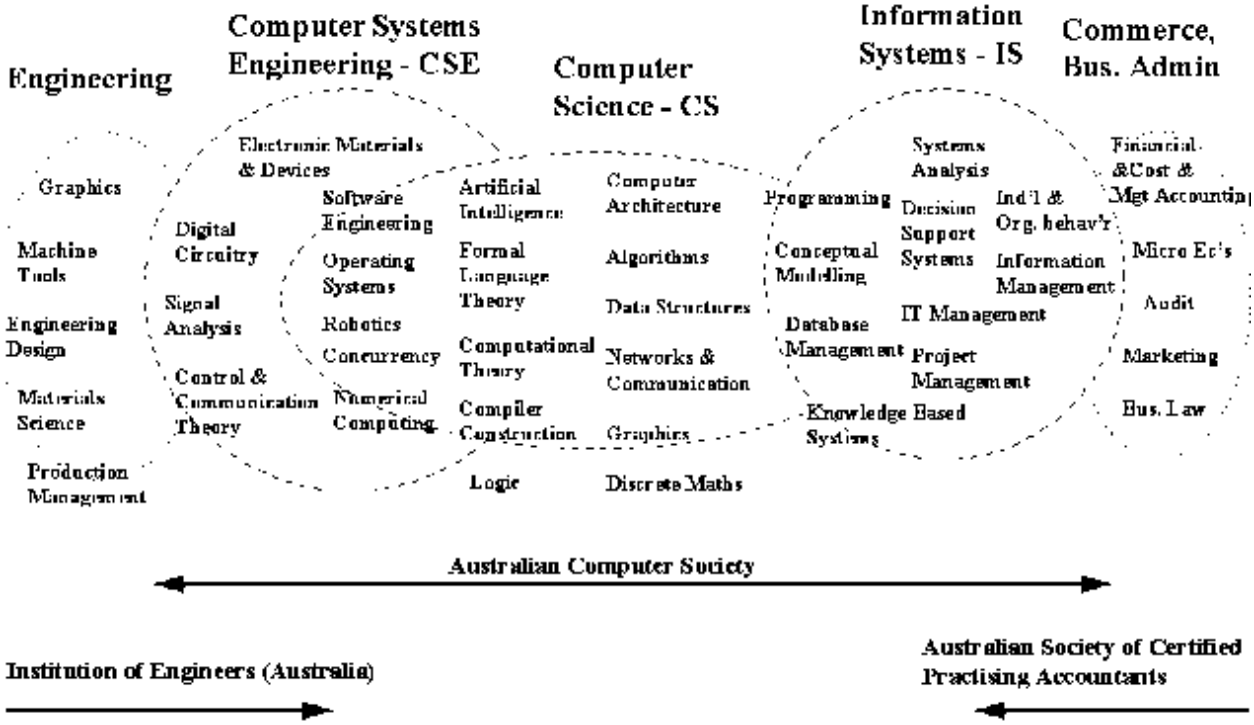


Figure 1: A Conceptual Model of I.T. Related Groups

The ACS BOK recognizes 15 curriculum areas, and defines requirements for the level of understanding of each area expected of professionals in the three main areas of the professional spectrum. The curriculum areas, and the expected achievements, are set out in Table 1.

Table 1 - Required Depth of Coverage

AREA OF KNOWLEDGE	Understand			Apply			Design		
	IS	**CS**	**CSE**	**IS**	**CS**	**CSE**	**IS**	**CS**	**CSE**
1. Computer Organization and Architecture	Y	Y	Y	Y	Y	Y			Y
2. Conceptual Modeling	Y	Y		Y			Y		
3. Database Management	Y	Y	Y	Y	Y		Y	Y	
4. Data Communications and Networks	Y	Y	Y	Y	Y	Y	Y	Y	Y
5. Data Structures and Algorithms	Y	Y	Y	Y	Y	Y		Y	Y
6. Discrete Mathematics	Y	Y	Y	Y	Y	Y	na	na	na
7. Ethics/Social Implications/ Professional Practice	Y	Y	Y	Y	Y	Y	na	na	na
8. Interpersonal Communications	Y	Y	Y	Y	Y	Y	na	na	na
9. Program Design and Implementation	Y	Y	Y	Y	Y	Y	Y	Y	Y
10. Project Management	Y	Y	Y	Y	Y	Y	Y	Y	Y
11. Information Security	Y	Y	Y	Y	Y	Y	Y	Y	Y
12. Software Engineering and Methodologies	Y	Y	Y	Y	Y	Y	Y	Y	Y
13. Software Quality Principles	Y	Y	Y	Y	Y	Y	Y	Y	Y
14. Systems Analysis and Design	Y	Y	Y	Y			Y		
15. Systems Software	Y	Y	Y	Y	Y	Y		Y	Y

Based upon this distribution, the Areas of Knowledge are classified into four groups, with the grouping of Interpersonal Communications, Ethics/Social Implications/ Professional Practice, Software Quality Principles, and Project Management being seen as common to all fields of IT. The domain of systems and software engineering also contains the areas of Data Structures and Algorithms, Program Design and Implementation, Software Engineering and Methodologies, and Information Security.

What conclusions can be drawn from the views of the professional societies? The principal one is that their views are currently in a state of flux! The second is that there still appears to be considerable difficulty in these forums to distinguish effectively between software engineering and computer science. The final point is that a number of issues seen as of substantial importance by the industry - notably, measurement and reuse - do not find explicit mention in these documents.

4. IT Education - the View of the Educators

The second route for examining the status of IT (and specifically SE) education is to examine the initiatives of the educators. Here, I must emphasize, we are looking at a small and perhaps unrepresentative sample of individuals and organization. Specifically, we focus mainly on those educators who have been prepared to discuss publicly the ways in which they try to teach software engineering.

I wish to focus on educational programmes that provide the initial professional training in information technology; so far as is possible, I will discuss programmes specific to software engineering; however, such courses are still very few. This focus means that I will not be addressing the principal educational opportunity for software engineers in the USA - the post-graduate (commonly Masters) programme. My reasons for this are simple; I believe that sufficient evidence exists that effective undergraduate programmes in software engineering are emerging internationally, and, this being so, market forces will inevitably led to their acceptance as the principal entry programme for the profession. A summary of other arguments on this issue can be found in the report by Ford and Gibbs.[10]

In the undergraduate domain, Ford and Gibbs list 13 degree courses in software engineering in the UK, and three in Australia; there is at least one additional course in Australia - at Griffith University - and there may well be more. In the USA, Gibbs and Ford refer to three programmes offering entry-level education focused on software engineering. They comment:

"Within the last ten years, most computer science curricula have added a one-semester elective course on software engineering. Such courses present software engineering in a superficial manner, comparable to an attempt to teach all of civil engineering or all of mechanical engineering in a single, one-semester course. Furthermore, being elective courses, not all students take them. Thus the graduates enter the profession with only bits and pieces of knowledge about software engineering."[10, p. 20].

In these software engineering programmes, the focus is generally on the issue of "programming in the large" - the application of the theory and practice of the matter of computer science to the problems of large system development. Given the context, it is not surprising that the primary focus of many of these "software engineering" programmes is more on the problems of project management than on the specific technical issues associated with large systems. Thus, factors such as software architectures, reuse, system evolution and maintenance tend to be overlooked. In courses with a larger concentration on engineering issues, a broader spread of technical issues is possible; thus, the software engineering programme within the Information Technology degree at Griffith University covers such issues as cleanroom software engineering; software process assessment and improvement; software architectures; and the elements of the personal software process, as well as project management, both as a course in its own right, and through the conduct of a year-long group project.

The use of a group project has tended to be a common and almost universal technique in software engineering education. The basic variants of group projects have been described by Shaw and Tomayko,[11] but some interesting approaches to course design and assessment have been introduced in recent years. Of particular note has been the use in several courses[12,13] of assessment approaches based upon process assessment models, particularly the SPICE approach.

In developing the project courses for our own degree programmes, we have employed a variety of approaches. In our basic degree - the Bachelor of Information Technology - all students do a one-year group project. A detailed Quality Manual[14] serves as the basis for the course, and is made available through the Web. The manual is structured on the basis of the international standard for software life cycle processes, and defines a set of standard processes that are to be implemented in the project. Students work in groups of between 4 and 6 students. A key feature of the course is that all of the projects are for "real" clients, drawn from industry, government and voluntary groups in the community, with only a small minority being in the nature of academic projects. Students are assessed on a wide range of criteria, with ample provision for the separate assessment of individuals within groups.

The project course has been a significant success. The primary learning experiences are in the areas of project planning, control, and requirements elicitation; the discipline of fully documenting a real project adds a significant element of realism to the degree. Clients from the software industry have commented that the standard of documentation achieved by the students is frequently in advance of their normal industry practice. Informal evaluations of process capability indicate that performance at level 2, approaching level 3 in some areas, can be achieved.

For our new four-year degree in software engineering, students will undertake the regular project course in the third year of the programme. In the final year of study, there is planned a second project course. For this, a different approach is planned; the "large group" model [11] will be used, with the whole class forming a single project team. This approach has not been possible in the B Inf Tech programme, where up to 130 students are enrolled in the project course; however, numbers in the B Soft Eng programme will be limited to a maximum of 20 per year. In this case, students will participate in several sub-teams over the course of the year. Assessment will be heavily based on the use of process assessment approaches, with ratings of process capability in different processes being used as one basis for assessment of team members. While this course is still in the planning stages, we are confident that this approach will provide significant learning experiences for the students, and equip them well for their future in the profession.

5. Conclusions

I have argued on several occasions[15] that our current approaches to teaching computer science - the basic educational experiences for our budding software engineers at present - in fact work against providing them with a firm basis for professional practice as software engineers. In key areas - peer reviews, reuse, and in some cases testing - the focus on individual work and assessment builds a cultural bias that such practices are not consistent with "real" software work. Humphrey comments:

"At root, current software development practices are nearer to a craft than an engineering discipline. The professionals have private techniques and practices which they have learned from their peers or through personal experience. Thus, few software engineers are aware of or consistently practice the best available methods. The initial PSP data show that very few engineers use such proven practices as disciplined design methods, design or code reviews, or defined testing procedures."

Project courses provide one mechanism whereby students can gain experience in the use of real industrial-type practices. A word of caution needs to be made, however; while there is near unanimity that project work in some form is an important way of delivering the types of skill we need in software engineers, there is not always a clear educational rationale as to why this is so. Too often, it is simply matter

of repeating some mantra regarding “industry exposure” or “group dynamics”. This is not to downgrade the importance of both of these issues; however, project courses can - and do - provide much more for the education of software engineers.

With the emergence of professional discipline, the need in the introductory educational experiences is for experiential learning oriented towards realistic industry practice. While it is important for the fundamentals of computer science to be incorporated into these programmes, there must in addition be the integration of theory into the construction of software-based systems. For this to succeed, there must be cooperation and collaboration with industry. Here we find the need for real input from industry into the efforts of academia. I endorse the views of Ford and Gibbs:

“The segment of the United States academic community that provides professional education is generally responsive to the needs of the professions they support. In particular, schools listen to the practitioners, the industries that employ them, and their professional societies. Thus the most important step in the establishment of initial professional education for software engineers is a clear expression of need from the software community. We believe that the companies who employ software engineers will have the loudest voice and that they need to exercise it.”[11, p 23]

My words to the information technology industry - those who will benefit from the availability of people with a professional approach to the development of software - is that they should do all in their power to increase the emphasis on engineering skills in computing degrees. They should collaborate with their local universities through the provision of meaningful and realistic projects, or offer studentship positions where these are needed. They should make clear to the designers of courses that they prefer graduates with key competencies required for the practice of engineering.

The move towards professional engineering practice in software development cannot now be reversed. The availability of personnel at the entry level to the profession with the necessary skills and competencies will help to accelerate this move, with long-term benefits for the profession and the industry as a whole.

References

1 Paulk, M.C., C.V. Weber, B. Curtis and M.B. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison Wesley, 1995.

2 El Emam, K., J-N. Drouin, and W. Melo, eds, *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE CS Press, 1997.

3 Humphrey, W.S., *A Discipline for Software Engineering*, Addison Wesley, 1995.

4 Boehm, B., “Seven Basic Principles of Software Engineering”, *J. Systems Softw.*, 3, 3 - 24, 1983.

5 McConnell, S., “Software’s Ten Essentials”, *IEEE Software*, Vol 14, No 2, 143- 144, April 1997.

6 “Jabir”, A Search for Fundamental Principles of Software Engineering, Report of a Workshop conducted at the *Forum on Software Engineering Standards Issues*, Montreal, October 1996; http://saturne.info.uqam.ca/ Labo_Recherche/Lrgl/ses96/report/jabir.htm

7 *The Institute, a news supplement to IEEE Spectrum*, July 1997, p1.

[8] The Task Force on Body of SE Knowledge, *Report on Analyses of Pilot Software Engineer Survey Data*, http://www.computer.org/tab/seprof/survey.htm, March 1997.

[9] Underwood, A., *The ACS Core Body of Knowledge for Information Technology Professionals*, Draft Version, July 1996. http://www.acs.org.au/national/pospaper/bokpt1.htm

[10] Ford, G., and N. Gibbs, *A Mature Profession of Software Engineering*, CMU/SEI-96-TR-004, Software Engineering Institute, Carnegie Mellon University, 1996.

[11] Shaw, M. and J. Tomayko, *Models for Undergraduate Project Courses in Software Engineering*, CMU/SEI-91-TR-10, Software Engineering Institute, Carnegie Mellon University, 1991.

[12] Veraart, V. and Wright, S., "Using SPICE as a Framework for Software Engineering Education - A Case Study" in *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, ed. K. El Emam, J-N. Drouin, and W. Melo. IEEE CS Press, 1997.

[13] Terry, J.E. and S. Hope, "The Simulation of Formal Management Reviews to Increase Quality in Undergraduate Software Engineering Projects", unpublished draft, 1997.

[14] Rout, T.P., *Quality Manual - SY13194, Information Technology Project*, School of Computing and Information Technology, Griffith University, http://www.cit.gu.edu.au/courses/undergrad/binftech/subjects/SY13194/

[15] Rout, T.P., "Quality, Culture and Education in Software Engineering", *Australian Computer Journal*, Vol 24, No 3, Aug 1992, pp. 86-91.

DISASTER RECOVERY IN DISTRIBUTED APPLICATIONS: A PRACTITIONER'S PERSPECTIVE

Philip J. Brown, CDRP
Senior Project Manager
Comdisco Professional Services
Atlanta, Georgia 30082

Robert F. Roggio, Ph.D.
Professor, Computer and Information Science
University of North Florida
Jacksonville, FL 32224

ABSTRACT

As developers and users of today's modern software systems, we are reasonably comfortable with issues of backup and recovery for mainframe applications. But what about true disasters? The real world provides us with too many instances of man-made and nature-caused phenomena, such as hazardous chemical spills, massive fires, major earthquakes, building sabotage, devastating tornadoes, and horrendous hurricanes most of which provide us no advance warning. These disasters transcend our notions of a limited backup and recovery and dramatically impact our abilities to provide vital, uninterrupted services to large and widely-dispersed customer bases. The inability to continue critical applications is exacerbated significantly with the ever-increasing number of distributed applications characterizing many modern application systems. No longer is the mainframe central to recovery, Now, data is distributed over wide geographic areas and processes are run on dissimilar end-user personal computers and workstations with non-standard interfaces. Homogeneity is gone. The diverse computing needs of a literate, modern society has forever changed the face of recovery from disasters. This paper addresses the notion of disaster recovery in distributed systems by presenting a brief look at the evolution of distributed applications , problems in recovering distributed systems from disaster, and solutions presenting many practical criteria and heuristics that practitioners may use in specifying software requirements.

KEYWORDS

Distributed Applications, Disaster Recovery, Business Continuity, Contingency Planning

BIOGRAPHY

Philip Brown has recently assumed the position of Senior Project Manager at Comdisco Professional Services based in Atlanta and is currently engaged in directing the contingency planning for Fortune 500 accounts. For several years prior to this, he served as manager of contingency planning at the JM Family Enterprises. In this capacity, he has accrued considerable experience in consulting with many companies in planning, assisting, and testing computer systems recovery vulnerabilities resulting from natural and man-made disasters. He has taken part in a number of actual recoveries. Mr. Brown is a member of the Association of Contigency Planners, and he is currently working on his thesis for his Master's Degree at The University of North Florida.

Robert Roggio is a professor of computer and information sciences and the University of North Florida in Jacksonville. His primary interests are in requirements analysis, software design, design methodologies and tools, software testing, and software quality issues. His background includes twenty years in the U.S. Air Force where he was involved with definition, design,

programming, implementation, and maintenance of standard Air Force applications data systems world-wide. In addition to professorial duties, he is currently international president of Upsilon Pi Epsilon, the computer science honor society, and a commissioner in the Computer Science Accreditation Commission (CSAC), the accreditation body for undergraduate computer science programs.

INTRODUCTION

As practitioners, we must expect the unexpected in our software applications: we specify requirements this way, we design this way, and we simply think this way. For many years, issues of backup and recovery have been integral to applications developed for mainframe computers. But mainframe computing has matured, and recovery is a feature with which most mainframe processing environments are comfortable.

The real world provides us many examples of risk: unanticipated hazardous chemical spills, massive fires, major earthquakes, building sabotage, devastating tornadoes, and similar nature-caused or man-made events for which there is no advance notice. When risks are mundane, people purchase insurance: auto, boat, plane. But system recovery from a disaster and resulting failure of critical applications can rapidly transform a corporation from a customer-satisfying, revenue-producing enterprise to one in serious litigation and possibly out of business. It is the need to provide quality uninterrupted services to a large customer base that has led to the creation of elaborate disaster recovery plans to minimize the loss due to computer system unavailability.

Companies purchase software, develop software, or contract with application support providers with little regard to the quality of the software relating to disaster recoverability. Except for special purpose systems in life support industries and similar businesses that have built-in fault tolerant capabilities necessary for mission-critical applications, the typical criteria checklist focuses on required functionality, software price, delivery date, and system performance. While these criteria may have sufficed in the past, the continued rapid growth of client-server applications with distributed processing using distributed data on heterogeneous platforms, concerns for recoverability must be broadened.

For mainframe applications, there is a single environment. One can take a snapshot, that can for the most part, be recreated on another equivalent environment. Supporting these needs is an extensive network of recovery vendors available to establish a processing base with the appropriate operating system, support software, applications, and data. The environment is rather simple; it is homogeneous. Backup and recovery in this environment constituted disaster recovery. Offsight tape backup storage and recovery was the norm. However, in today's growing complex distributed systems, traditional "backup and recovery" constitutes only a subset of a more comprehensive disaster recovery paradigm (see figure 1).

Recovery from a disaster in the distributed environment presents an entirely new set of parameters for software practitioners. One of the many obstacles to a successful recovery is the practical impossibility of having available sheer numbers of like servers and devices necessary for a total recovery. Each server presents its own unique blend of environmental constraints and may require varying technical support expertise. In the client-server environment, one typically finds modifications to the original vanilla device at the time of installation plus various performance tuning efforts made since the device has entered service. Specialized technical support expertise typically boils down to a precious few technicians who understand how the devices have been made to work in the production network. Documentation that would enable a technically competent person to recreate the device does not stay current - if it exists at all.

In the event of disaster, operations must frequently be moved to an alternative site(s). The cost of

maintaining or contracting for these redundant/backup services is very high. Today's device requirements do not meet tomorrow's device disaster recovery needs. Consolidating applications from different servers onto one or more servers introduces a new set of challenges. Some of the more significant problems include location and quality of the restored data, end user access paths, end user security access, different resource requirements by the applications, different configuration requirements for the applications, different configuration requirements for the server, different operating systems, different network operating systems, and more.

So, what does the software practitioner do about this? How can recoverability in a distributed environment be brought into the comfort zone for those responsible for viability of distributed applications? Just as software testing, once thought of as a separate phase in traditional software development paradigms, is now an integral aspect threading its way through all phases of development, so too disaster recovery must be addressed through all phases of the software development life cycle. It is one thing to cite in the specifications that recovery from disaster must be accommodated. But it is quite another to clearly and unambiguously define the extent of recovery required, place real world constraints (bound) around it, design for it, program for it, and design tests for it - while recognizing the non-homogeneity of distributed environments.

The authors recommend enhanced treatment for disaster recovery in all major application documents starting with clear and unambiguous treatment in the specification document, design, coding, and test plan documents. All maintenance manuals, operator manuals and user manuals must address these critical issues.

ELEMENTS OF SOFTWARE QUALITY

Software Metrics is a field of software engineering that (among other things) attempts to "measure" the software experience - both process and product. One of the most popular metrics is the "size metric,"

Figure 1. Disaster Recovery showing Backup and Recovery as a Subset

whereby software developers attempt to measure the size of a product. Lines of code (LOC) (certainly not a reliable metric when used in isolation) and function points are among popular metrics for assessing progress in a development schedule.

Other metrics indicating software quality are often used during the operations phase, such as mean time between failure, mean time to repair, and relative impact of the failure. For example, one error may be cosmetic - an error for sure, but resulting in no serious system malfunction; another may be extremely unlikely to occur, but its occurrence may bring the system to a halt.

According to Shach [2], many metrics may fall into categories such as product metrics (measuring some aspect of the product itself such as its size or reliability) and process metrics (used by

developers to measure and improve their software process, such as how a development team discovers, reacts, and learns from errors found and corrected during development.)

Software quality assurance (SQA) seeks to ensure the software product is correct and the software process is a viable one. Our guiding principle for verification and validation is encapsulated with the words of Barry Boehm [7] who succinctly defined verification and validation in the following way: Verification: Are we building the product right? And Validation: Are we building the right product. The software product is "verified" at the end of each phase and "validated" at the completion of software development to ensure requirements have been satisfied. The software process is improved by evaluating the development process, measuring the efficacy of various review processes (e.g. design, code, management), estimating the impacts of management decisions undertaken during software development (e.g. adding programmers or analysts), weighing the value of tools used (e.g. CASE, workbenches, statistical quality control measures and tools), and efforts at continuous process improvement. Certainly there are comprehensive standards that may be used to guide and measure our own internal processes: ISO 9001 and the Capability Maturity Models (CMM). [5]

Interestingly, the most popular software engineering books from which many of tomorrow's software developers are taught give little or no mention of disaster recovery and only few words to backup and recovery. Pressman [1] cites that "...computer-based systems must recover from faults and resume processing within a pre-specified time." Further, he cites that "...processing faults must not cause overall system function to cease." He concludes with, "If recovery requires human intervention, the mean time to repair is evaluated to determine whether it is within acceptable limits." [1] No mention is made of the more inclusive and far-reaching: Disaster Recovery. What then is Disaster Recovery and how it relates to Software Quality?

"Quality" may be defined in many ways, but generally it is defined as "conformance to requirements" [3] or, "quality must be defined as conformance to requirements, not as goodness" [4] and similar definitions. The common descriptor is conformance to requirements. Certainly quality must include recoverability. Yet in reviewing several highly regarded books on quality software management and quality assurance, topics such as recoverability and disaster preparedness are noticeably absent.

But what about "disaster?" What about the hurricane that destroys a corporation's data center? A tornado? Flood? Earthquake? Sabotage? Every location in the United States has some history of natural disaster recorded. Some of these, such as hurricanes, do not arrive unnoticed; others do. Recovery from natural and man-made disasters must be assured.

When one views modern day applications with massive distributed processing and widely-distributed databases across multiple heterogeneous processing environments interfacing with untold numbers of unlike client workstations, the process of recovering anything but trivial applications from a true disaster can become daunting indeed!!

THE NEW LEGACY - DISTRIBUTED APPLICATIONS

The universities of the late 80's and early 90's were fertile soil for rumors of the imminent demise of the mainframe. The curriculum and student interest reflected this belief. As these students found their way into the work force they were eager to write PC based applications but found a lack of serious commitment to their development. The PC applications that did survive typically have been supported by the developers. Useful applications made the developers the new data processing stars. In a fraction of the time it took to develop an application for the mainframe, a PC application was ready for use. The productive, or fun, applications were loaded on many machines. Sharing data presented momentary problems. Application developers extended their expertise to include networking the PC's together.

Phenomenal growth in PC based third party software, affordable hardware, and advances in networking have all contributed to the PC revolution. Corporate end users have become knowledge and information brokers. Software and hardware vendors market their products directly to the end users bypassing the data processing department's rules and regulations. High powered hardware empowered by sophisticated software often enables companies to gain a competitive advantage. The new hardware and software have become the new legacy systems. Many company's existence now depends on PC applications. Recovery requirements now extend to the distributed environment, and hence the server containing the application and its data must be backed up and recovery ensured.

DISTRIBUTED RECOVERY DILEMMA - THE PROBLEM SPACE

There are many challenges in planning for and actuating a recovery from a disaster in a distributed environment. A few of the more significant ones are: proliferation of LAN's, changing hardware and software, lack of standards, lack of change controls, poor backup strategies, lack of network resources, big iron recovery mentality by senior management, mainframe-centric recovery vendors, and applications not designed for recovery.

The Proliferation of LANs. The proliferation of LANs, often outside the knowledge or control of the data processing department, has provided the highway within the corporation to share information and data. Difficulties have arisen when departmental LANs have needed to communicate with each other. Differences have abounded in how the LAN has been installed, how servers have been configured, and who has been tasked with making the LANs communicate. The early satisfaction gained by connecting departmental PC's together has given way to rising frustrations in meeting the growing expectations of the end users. Few if any companies have a LAN/application roll out strategy. There has been barely enough time to get the LAN's connected and communicating. Documentation outlining the rationale behind the current configuration of the distributed architectures is not misplaced; it has rarely existed. How then can a recovery be undertaken?

Changing Hardware and Software. It seems as if new hardware and software is introduced into the distributed environment on a daily if not hourly basis. End users are hungry for the next release of existing applications or new applications that will make them more productive or facilitate their current tasking. Support personnel apply fixes downloaded off the Internet that change the distributed environments configuration to resolve production problems or improve performance. Peripheral services are added or enhanced through the installation of new hardware. Network management is enhanced by loading agents on workstations, servers, and other network devices. Vendors provide trial products directly to the end user that find their way into the distributed environment adding functionality with little regard to recoverability.

Lack of Standards. It is impossible to develop standards in an environment that changes more often than the weather. Reasonable standards require research and analysis. A standard must be striving towards a greater goal than restricting accessibility to productivity tools. How can a standard strive towards an undefined goal? A standard that is not matched with immediate or near future benefits is destined to be ignored. The resources to perform the research and analysis that will allow companies to plan their way out of their dilemma are directed to more immediate problems or opportunities rather than facilitating future recovery. Without standards, chaos is lurking under the covers of the distributed environments.

Lack of Effective Change Control. Controlling the inflow of new hardware and software has clear hurdles. The points of entry for any type of change to a distributed environment are virtually unlimited. Small insignificant changes that are not recorded or remembered add to the complexity of the environment and thus complicating recovery. Control and personal computers are

oxymoron. The subtle attempt by many corporations to refer to PC's as workstations is often met with contempt. Individuality is personified by the PC.

Poor Backup Strategies. The prevailing backup strategy in the distributed environment closely parallels the early backup strategy in the mainframe environment, namely full volume backups. The same issues plus many new ones that forced a migration away from full volume backups on the mainframe are waiting to be addressed in the distributed environment. The easiest one to recognize is the time constraints within the nightly batch window. Two factors come into play here: the size of the server for backup and the available time to perform the backup. In two years the standard file server's hard drive has increased from 2 Gig to 20 Gig - a 1,000% growth rate in two years. In the same two years the available time to conduct backups has shrunk from eight hours to four hours. Extended hours of operations and more nightly batch processing performed on the server are the culprits. There is reason to believe this trend will continue.

Significant constraints limit the number of options in developing an effective backup strategy that can support or facilitate a recovery from a disaster. The most serious one is the lack of data storage management in the distributed environment. Because hard drives are inexpensive, it seems that few are concerned about how the operating system, applications, and data are segregated. The time has come for companies to realize that data storage management in the distributed environment is a necessity. Developing standards that segregate the data, applications, operating system, and imposing controls to enforce the standards is needed for the distributed environment.

Lack of Network Resources. Data processing resources are in great demand. Few companies enjoy the luxury of having a data processing department dedicated to developing long term strategies for the distributed environment. What is long term in the distributed environment - six months, one year, etc.? How can a company justify allocating a highly skilled employee to developing strategies when the immediate demands consume the hours in the day and the days in the week? How many highly skilled employees are enticed into forging a career path in strategic planning? How many companies have enough highly skilled employees to staff the back logged to-do list? Thus few, talented individuals are involved in strategic recovery of systems.

Big Iron Disaster Recovery Mentality. Disaster recovery has its roots in the mainframe. Most medium to large size companies have contracts with recovery vendors for access to computer equipment for recovery testing or disaster declaration. Monthly expenses for the contracts commonly are in the tens of thousands with a growing number of companies paying six figures a month. The traditional contract covers a mainframe configured with enough memory, cache, DASD, tape drives, communication controls, inbound/outbound communication links, printers, and dumb terminals. Data processing departments budget anywhere from one to five percent for disaster recovery. Senior management feels secure the company can recovery from a disaster. After all, the company pays that huge monthly premium (as insurance). Certainly when a disaster strikes, the data processing department will recover the mainframe and everything will be fine. What senior management has been slow to recognize is that no one accesses the mainframe through a dumb terminal anymore. Further, today's work force is dependent upon distributed applications and resources to perform even the most fundamental elements of their jobs. A frequent comment by senior management that the employees can always go back to the way it used to be done is not valid either. The knowledge base is gone and the business has changed. Many companies are poised to recover their mainframe but still face the likelihood of going out of business because end-users can no longer access the applications.

Mainframe-Centric Recovery Vendors. Recovery vendors command significant costs for what amounts to time sharing on data processing equipment. The recovery vendor provides climate controlled facilities, computer equipment, and communication links. Recovery contracts typically extend from three to five years. The longer the terms of the contract the more attractive the monthly charges. Upgrades to contracts are an option for companies attempting to keep their recovery environment current but this is expensive. The better recovery vendors offer technical

expertise on the platforms in their product mix. The mainframe and its peripherals generate the lions share of proceeds. This arrangement has served the recovery industry very well.

Recovery vendors are moving into the distributed environment in response to their customers demands. Several work group recovery centers have been developed over the last three years. The centers were initially populated with a basic configuration comprised of a workstation with a LAN-attached PC and a telephone. Companies contracted for a specific number of workstations, file servers, communication lines, and switchable links back to their corporate WAN's. Problems came about for the recovery vendors as soon as they began marketing the work group recovery center. Each company has different distributed recovery equipment requirements. The only viable solution for the recovery vendor was to wait until a customer was willing to pay the cost to upgrade the current equipment or pay to bring in new equipment. Once the new or upgraded equipment was available, the recovery vendor adds the equipment to their product offering.

Companies can plan their mainframe migrations to new equipment and amend their recovery contract to reflect the growth. The distributed environment is scaleable and very responsive to changes in business. This presents significant challenges for recovery planners. How do we contract for recovery equipment in the distributed environment.? Today's standards have a serviceable life of perhaps 18 months. Yet the average recovery contract is from three to five years. Upgrading the equipment is always an option. Unfortunately the recovery vendors can not afford to stay ahead of the wave of equipment change. Attempting to anticipate their customers needs can prove disastrous for a recovery vendor because they could purchase hardware that no company will contract for.

Applications Not Designed for Recovery. Many applications are not well suited for a distributed recovery. Applications have been designed for an initial installation and a life serving its users. Applications often find themselves being used in ways never intended by their developers. These same applications may have frequently become popular with users spread across time zones. When this happens, recovery planners must address a new set of questions. An example will help illustrate.

> A software package allows a company to electronically store documents. The company can make changes to the document online to reflect legal, regulatory, or business changes as needed. The software saves the company the cost of inventorying forms and keeping the forms current. Now when a document needs to be sent to a customer, the form is brought up online, pertinent customer information is added, and the form is sent to the printer or faxed. The company soon developed a front end application to the software package. The application allowed the end user to select an electronic document from a pick list and select a customer from a database. The application then retrieved the customer information from the mainframe, entered the pertinent customer information into the document, and sent the document to the printer.
>
> The combination of purchased software package and in-house developed application gained wide acceptance. The software package and the electronically stored documents were distributed to several other locations, but the database remained in the originally installed location. A small program added to the remote client's PC allowed the user to select an electronic document from a localized pick list and select a customer from the remote database. The application, now distributed throughout the corporation, performs very well. The network intensive movement of the electronically stored document is localized to the LAN supporting the end-user. The request for and retrieval of customer data is light network traffic and travels over the corporate WAN. This is a well designed and implemented application. However, it presents disaster recovery challenges.
>
> If the center that houses the database supporting the software package and developed application became inoperable, recovery of the database is not the most important

recovery requirement for the center. The base infrastructure and any revenue-generating applications are assigned a much higher recovery priority. Interestingly, however, the other locations that need to access the database are not in a recovery mode and their daily productivity depends on, in part, access to the database. Consideration, therefore, should be given to moving the database recovery priority up to the same level as the revenue-producing applications. The application was not designed for recovery yet must now be counted as an application required during a disaster recovery.

DISTRIBUTED RECOVERY DILEMMA - A CURRENT APPROACH

We have thus far presented a list of actual events experienced by the authors in attempting to recovery critical applications in failed distributed environments. While the authors feel that the future solution to software recoverability in distributed systems is found in specifying and designing recoverability into software (as outlined in the next section), the following materials represent the current state-of-the-art contingency planning framework for addressing distributed recovery.

Support for the Business Continuity Plan (BCP). No longer can we hope to recreate the entire data processing environment as was the plan under traditional disaster recovery. Unlike the mainframe where a recovery constitutes the entire machine, only strategic pieces of the distributed environment can be recovered in a short period of time. Recovery planners must develop a different paradigm. A move is necessary from a data-processing-centric world to a business process view. The notion of focusing recovery efforts on mitigating loss and continuing those processes that will keep the patient alive was formalized as "Business Continuity." Business Continuity Planning (BCP) has gained rapid awareness and acceptance. 'According to a recent *Contingency Planning & Management*/Ernst & Young LLP study, [6] 95% of companies surveyed are either developing or have some type of BCP in place'. A number of the key issues driving this growth is provided in Figure 2.

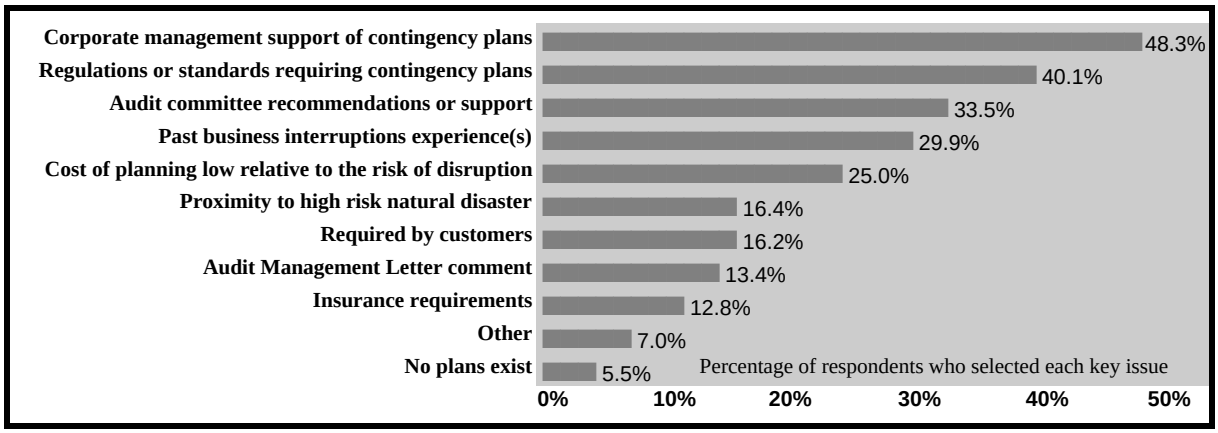


Figure 2. What key issue(s) initiated the development of a BCP?

Causes of Business Interruptions. The types of business interruptions for which companies build contingency plans varies widely. Figure 3 [6] identifies the most common type of business interruptions companies have experienced over the last five years. Certainly degrees of interruptions must be considered and not every interruption requires a contingency plan. The most common interruption is a power outage. Does it make sense to build a contingency plan for a five minute power outage? How about a five hour power outage? Or a five day power outage? There are many factors involved in determining what should be planned for.

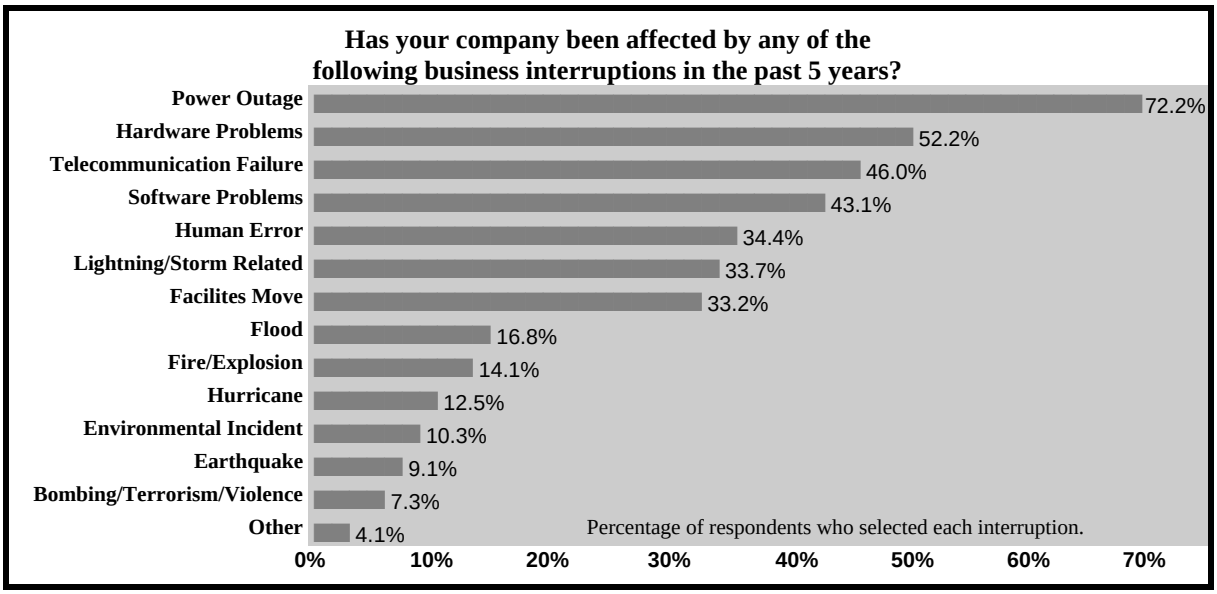


Figure 3. Type of Business Interruptions

Recovering the Critical Applications. At the heart of any business processes that utilizes computing systems is the application. If the distributed environment can not be recovered completely (and this must be anticipated), then the litmus test of what is required for business process recovery is the application(s) that enable the overall business process. Identification of the applications supporting the critical business process is the fulcrum. On one side is a business process with a list of one to many applications that must be available. On the data processing recovery side is a focus on a segregated recovery.

Selective Recovery: Business Impact Analysis (BIA). A significant amount of data gathering, analysis, and planning/partitioning/prioritizing is required to determine which parts of the distributed environment should be recovered. Contingency planners have developed a Business Impact Analysis (BIA) methodology to scope the recovery efforts to those functions critical to the immediate recovery of a business. The audience for the BIA range from department heads to front line management. Key components of a BIA include questions concerning the operational, financial, legal, and regulatory consequences to the business if an interruption occurs. Additional components of the BIA are concerned with the time of the interruption and the length of the interruption. The BIA provides qualified data. Analysis of the data identifies the base data processing infrastructure needed for a distributed environment recovery and a sequence for application recovery from a disaster. Sample BIA questions follow.

The operational impact questions are concerned with the loss of productivity resulting from performing a function manually (if possible) as opposed to an automated process. Key issues are:

- ✔ What are the major objectives of your department?
- ✔ How are these objectives achieved?
- ✔ How would these objectives be impacted by _____?
- ✔ How would you perform _____ if _____ happened?

Figure 4. Operational Impact Questions

Financial impact questions focus on the revenue streams produced or supported.

- ✔ How would a business interruption affect cash flow?
- ✔ How long would working capital last during an interruption?
- ✔ What is your current market share?
- ✔ How long will your customers remain loyal?
- ✔ What options do your customers have?

Figure 5. Financial Impact Questions

Legal and regulatory impact questions draw attention to external factors that must be considered. Litigation can shut a business down as quickly as the loss of a key revenue stream. Potential sources of litigation must be considered.

- ✔ Are there contractual obligations to provide products or services?
- ✔ Do you have Service Level Agreements with your customers?
- ✔ What recourse do your customers have if _____ happens?
- ✔ Do you have to comply with federal regulations?
- ✔ What kind of penalties can be assessed?
- ✔ Are you vulnerable to a lawsuit?

Figure 6. Legal and Regulatory Impact Questions

Time of the loss questions assist in prioritizing a recovery taking into consideration the normal fluctuation in business cycles.

- ✔ What would be the worst time for a business interruption?
- ✔ Is the worst time the same for all functions within your department?
- ✔ Have you experienced a business interruption?
- ✔ If so, what happened - what would you do different?

Figure 7. Time of the Loss Questions

Duration of the loss questions also assist in prioritizing a recovery. Most processes can absorb a minimal amount of down time.

- ✔ Under the Best/Worst Circumstances
- ✔ How long can you survive a 25% reduction in productivity?
- ✔ How long can you survive a 50% reduction in productivity?
- ✔ How long can .....?

Figure 8. Duration of the Loss Questions

The results of the BIA becomes the starting point in understanding and addressing data processing recovery requirements. Each business process required during a disaster recovery is examined to determine what applications are needed. For each business process, end-user,

application support, workstation support, database support, and network support personnel are interviewed (the groupings listed are generalized to reflect a logical segregation of the areas called upon to rebuild a distributed environment). A series of sample questions for each of these particular groups follows that helps direct the information gathering process.

The end-users know which applications get the job done. This paper focused on data processing topics but it is worth noting that most applications supporting business processes have requirements for resources or materials that are not in the data processing arena. A few examples are support manuals, mail handling, customer support, technical support, and banking relationships. For each business process the end-user is involved with, the same questions are asked. It is a good practice to review the results of an interview with a second end-user. It is common for one end-user to use only a portion of an application thus potentially leaving out key pieces.

- ✔ What applications do you use to complete the business process?
- ✔ What are the application inputs? Where do the inputs come from?
- ✔ What are the application outputs? Where do the outputs go?
- ✔ Does the application require 3rd party software?
- ✔ Do any events on the application trigger other processes?
- ✔ Are other applications dependent on outputs from the application?
- ✔ What test data can be used to validate a recovery?

Figure 9. Questions for End-Users

Application support involvement in a business process based recovery is critical. The daily performance monitoring and problem resolutions performed by application support lead to substantial changes to an installed application over time. Extensions to the intended use of an application are very common. Experience with applying fixes and new releases to the application provide insight on potential problems waiting for a re-installation or recovery of the application. Application support should know and may have diagrams of how the application interfaces with other applications and systems. Another key element application support can provide are names and numbers of key vendor personnel, external contractors, and peers in application user group who may be able to help resolve issues.

- ✔ Where is the application loaded?
- ✔ Who are the application users? (local, remote, external)
- ✔ Does the application access a database?
- ✔ Do other files or data sets support this application?
- ✔ What are the application inputs? Where do the inputs come from?
- ✔ What are the application outputs? Where do the outputs go?
- ✔ Does the application use 3rd party software (WP, Excel, Notes, etc.)?
- ✔ Do any events on the application trigger other processes?
- ✔ Are other applications dependent on outputs from this application?
- ✔ How often does the application change?
- ✔ Does the application need to be backed up? Media? Frequency?
- ✔ How will the application be restored? Rebuilt from CD, disk, or backup tape?
- ✔ What configuration files are needed to access the application?
- ✔ Is there application security?
- ✔ Is there an application security administrator? Who is it?
- ✔ Is an application license tied to the server or CPU?
- ✔ How can the application recovery be validated? Is test data required?

Figure 10. Application Support Questions

Workstation support builds the access to the applications for the end users. What specifics about how the workstation is configured needs to be known to effect a workstation recovery? What is the lowest common denominator about the configuration of 25 or 50 or 100 workstations that have to be recovered in fewer than 4 hours?

- ✔ If a workstation was lost who would be asked to recover the desktop?
- ✔ Which configuration files are needed to recover the workstation and do you have a copy of them?
- ✔ What installation software would be required to recover the workstation and do you have a copy of them?
- ✔ Are these configuration files different for different workstation operating systems?
- ✔ Are these configuration files different for different network operating systems?
- ✔ Is there a standard workstation image for the different operating systems?
- ✔ How would the workstation be recovered?
- ✔ How would you validate the workstation recovery?
- ✔ How long would it take to recover a workstation?
- ✔ How long would it take to recover all the workstation you are responsible for?

Figure 11. Workstation Support Questions

Database support is responsible for an application's data integrity (when an application uses a database). An application is not successfully restored until the database supporting the application is recovered.

- ✔ Who provides database support ?
- ✔ What kind of database supports the application? (Gupta, Sequel Server, Oracle, DB2, etc.)
- ✔ Where is the database located?
- ✔ Who are the database users?
- ✔ How is the database backed up? Media? Frequency?
- ✔ How will the database be restored? Rebuilt from CD, disk, or backup tape?
- ✔ Who will restore the database?
- ✔ How is the database validated?

Figure 12. Database Support Questions

Network support typically has the lead in recovering the distributed environment. The information gathered from the other functional areas provides network services with the information necessary for the reconstruction of a LAN.

- ✔ How many segments are in the network?
- ✔ Who manages the segments?
- ✔ Are there unmanaged segments?
- ✔ Are there current diagrams of the network?
- ✔ How many servers are there in the network?
- ✔ How are servers added to the network?
- ✔ Are the servers backup?
- ✔ If so what is the backup frequency?
- ✔ How many cycles of the backups are keep?
- ✔ Where are the backups stored?
- ✔ How would you recover a server?
- ✔ How would you recover multiple servers?
- ✔ If the servers had to be replaced, how long would it take to get replacement servers?
- ✔ Are any pre-arranged agreements in place with vendors to provide equipment in the event of a disaster?
- ✔ What other intelligent devices are on the segments? (gateways, bridges, routers, hubs, concentrators)
- ✔ Who is responsible for communications with mid-range or mainframes?
- ✔ Who is responsible for long line service?

Figure 13. Network Support Questions

CHALLENGE TO SOFTWARE DEVELOPERS

Much attention has been spent on outlining and defining key issues of software quality, the recovery issues created by a labyrinth of complex interdependencies common in today's distributed environment, and current attempts to identify the multiplicity of resources and personnel that must be brought to bear to recover a segregated, distributed system. Certainly these materials give us heightened awareness of the vulnerabilities and the associated risks of disaster recovery in a distributed environment.

What specifically is the challenge to software developers? The authors feel that one cannot address a future solution without a careful examination of the composite problem - namely, that software is developed for delivery and not recoverability. Thus the many issues and attendant questions presented are those that experience has shown arise when recoverability is planned or implemented. Shouldn't these questions be applied to various phases of software development? Some clearly apply to requirements gathering and specification; others are critical issues during design and implementation. Thus we are advocating that software developers explicitly address recoverability issues as an integral part of each phase of the software development process. Specifically, the authors feel that central element to viable distributed recovery lies in the following questions: how will the application be restored; how will it be rebuilt; how is the data preserved/recovered?

The problem is further rooted in our educational system. All standard textbooks addressing Software Development and Software Engineering address software life cycle events. Regardless of the development paradigm (structured design, object-oriented, or others) the generic development phases include Requirements Gathering, Specification, Design, Programming, Testing, Implementation, and Maintenance. None of these treat disaster recovery with anything more than a passing comment or vague reference citing that "this should be done." Formats outlining deliverables culminating requirements specification or software design barely mention disaster recovery - if at all. There is no comprehensive treatment of this subject taught in standard software engineering courses.

Let's look at one set of typical questions previously posed that affects Workstation support (see figure 11) and reexamine the impact of including these questions in software development. The second question asks, "which configuration files are needed to recover the workstation and do you have a copy of them?" In the installation of the client software, these files are typically loaded from a client-installation diskette. In a recovery, where more than 50 workstations must be recovered in less than one hour, this method is unacceptable because of the length of time each workstation would require for recovery. An acceptable option constitutes identifying which configuration files are required for the applications of the desktop and building a model configuration desktop on the file server. During recovery, the configuration files on the server can be copied to the workstations en masse thus requiring only a fraction of the time spent at each workstation. Clearly a better option. However, what if a configuration file statement for an application was missed? Why leave the recoverability of an application to chance? Rather than workstation support's best effort to create a model configuration on the server for application recovery, what if the application developer built a method to mass install the client piece from a server as part of the software development process? The authors assert that this is the best approach and postulate that this should be part of the requirements document.

Recoverability issues such as these, then, must form a critical component of the requirements document and their inclusion must be verified. The ability to reinstall the software and recover the data must be designed into the system, coded into the system, and incorporated into system test scenarios at validation time. Further, to ensure recovery compliance, the application must be installed, executed creating data, and be recovered on dissimilar hardware. Only then can the recoverability of the application and its data be assured.

References:

1. Roger Pressman, Software Engineering Principles and Practices, 3rd edition, 1992.

2. Stephen R. Schach, Classical and Object-Oriented Software Engineering, Third Edition, Irwin, 1996

3. Gerald M, Weinberg, Quality Software Management, Volume 1 - Systems Thinking, Dorsett House, 1992

4. Vincent, Waters, Sinclair, Software Quality Assurance, Volume 1, Prentice Hall, 1987

5. Bamford and W.J. Deibler, II, "Comparing, Contrasting ISO 9001 and the SEI Capability Maturity Model," IEEE Computer, Volume 26, October 1993, pp 68-70.

6. *Contingency Planning and Management,* Ernst and Young article on BCP - by Janis Keating with interviewing assistance from Jennifer Kline and Michelle Simonelli, April 1997, Vol II No 4.

7. Verifying and Validating Software Requirements and Design Specifications, B.W. Boehm, IEEE Software, January 1984, pp. 75-88.

OBJECT TECHNOLOGY ADOPTION -- A RISK MANAGEMENT PERSPECTIVE

Peter Hantos, Ph.D.

Xerox Corporation
701 South Aviation Boulevard, MS: ESAE-375
El Segundo CA 90245
Phone: (310) 333 - 9038
Internet: phantos@es.xerox.com

Sujoe Joseph

Carnegie Mellon University
Heinz School of Public Policy and Management
Pittsburgh, PA 15213-3890
Phone: (412) 268 - 4005
Internet: sujoe@andrew.cmu.edu

ABSTRACT

In this paper the authors provide a risk management perspective for large software projects adopting Object Technology. Many companies are looking to Object Technology as a means to achieve their strategic business objectives. Object Technology's promises are the ability to build complex systems of superior quality, with reduced development time and costs, and also to provide long term benefits such as maintainability, reusability and extensibility.

The authors carried out several software risk evaluations within Xerox and other companies. The method they have used for conducting the evaluations was developed at the Software Engineering Institute, Carnegie Mellon University. They found that, although Object Technology provides many exciting opportunities, projects adopting it carry some inherent risks that must be actively managed and mitigated. Projects -- particularly large ones with significant financial and technical commitments -- face challenges in many technical and management areas, but the most common risks, which this paper is concentrating on, are in the areas of new technology, staff experience, software development processes and system architecture.

The published experiences should be of benefit to first time OT adapters, and, at a minimum, provide increased awareness about the common risks of this relatively new field.

KEYWORDS

Risk Management, SEI, SRE, Object Technology

BIOGRAPHIES

Peter Hantos is Principal Scientist of the Xerox Corporate Software Engineering Center. His tasks include the coaching and mentoring of software process improvement teams across Xerox on software technology related issues, identifying and sharing best practices. One of his current responsibilities is the internalization and institutionalization of a systematic software risk management approach for Xerox. In his previous position, he managed a software engineering organization providing methodology, tools and infrastructure support for members of the Xerox Corporate Research and Technology Division located in El Segundo, California. Dr. Hantos is a Senior Member of the IEEE and Member, ACM.

Sujo Joseph is currently the Technical Director for Computing and Telecommunications at the Heinz School of Public Policy and Management, Carnegie Mellon University. His contributions to this paper are based on his earlier work at the Software Engineering Institute (SEI), Carnegie Mellon University. While with the SEI, he has conducted many risk evaluations of software-intensive systems, and co-authored an SEI technical report on the Software Risk Evaluation method. Before joining Carnegie Mellon University, he was a software engineer with Bull Worldwide Information Systems. Mr. Joseph is a Member of the IEEE and ACM.

© 1997 Xerox Corporation

INTRODUCTION

Today many companies developing software are looking to object technology [1] as a means to achieve their strategic business objectives. They are betting on OT to build complex systems of superior quality with reduced development time and costs, while providing long term benefits such as maintainability, reusability, and extensibility. OT, however, is a relatively new discipline that is continuously changing and rapidly evolving. Although OT provides many opportunities, projects adopting it have inherent risks that must be actively managed and mitigated to avoid catastrophes. Projects -- particularly large ones with significant financial and technical commitments -- face special challenges. In this paper we provide a risk management perspective for projects adopting OT. The authors of this paper have conducted several software risk evaluations within Xerox and other companies. We believe our experiences will be of benefit to large projects adopting OT for the first time, and at a minimum, provide increased awareness about some of the common risks.

Why did we direct our focus on OT? Aren't the same risks associated with any new technology? To answer this question we have to realize, that with respect to ***scope***, ***complexity*** and ***depth*** OT is unique. Expanding the somewhat broad-brush definition of OT in [1], we have to realize, that the ***scope*** of the OT adoption can cover the following technical areas: System Modeling and Simulation, Analysis, Design, Programming and Automated Code-Generation, Reuse, Distributed Systems, Middleware, Development/Debugging and Test tools, etc. It is clear that the decision is not simply whether to use the new technology or not. Do we have to tackle all the technical areas to be successful? If not, then how do we decide on the priorities and the order of implementation? With respect to ***complexity***, any of the mentioned technical areas can be very complex on their own, but we are dealing with numerous, highly inter-dependent new concepts as well, and proceeding without the understanding of these interdependencies can be very dangerous. Finally, looking at the question of ***depth***, after we decided on the areas to work on, we still have to decide how far to go while implementing these new concepts, and how rigorous we have to be to achieve success.

Software Risk Evaluation

Software Risk Evaluation (SRE) is a method developed at Carnegie Mellon University's Software Engineering Institute and is used for identifying and mitigating risks in software development projects [2]. It is based on a continuous risk management paradigm and a taxonomy [3] for identifying and analyzing software risks. Typically, five or six interviews are held across the technical and management functions of a project, during which issues are openly discussed and recorded on a flip-chart using the original wording of the person who brought it up. At the end of each interview session, issues are analyzed in terms of their impact to the project and their probability of occurrence.

After all interviews have been completed, the SRE team consolidates the risks within each category and presents the highest priority risks to the project members. Later the SRE team prepares a preliminary report of the risks and its context to the project's management. The preliminary report also contains suggestions for risk mitigation. Project management selects areas for risk mitigation and identifies

---

[1] In this paper the term object technology (OT) means object-oriented development processes and methods, object related standards, and associated products and tools from third party vendors.

[2] S. Joseph and F. Sisti, Software Risk Evaluation Method, Version 1.0, Technical Report, CMU/SEI-94-TR-19, December 1994.

[3] M. Carr et al., Taxonomy-Based Risk Identification, Technical Report, CMU/SEI-93-TR-6, June 1993.

individuals who can work with the SRE team to develop mitigation strategies. Typically, four or five mitigation area working sessions are conducted during which the SRE team and project individuals jointly develop strategies for risk mitigation. Details for implementing each mitigation strategy are also presented, including some high-level activities, tasks and rough estimates of the effort and time required to complete them. The results of the risk mitigation sessions are presented to the project management in a final report. For a review of the SRE approach from the users' perspective, see [4] in the last year's conference proceedings.

OT vs. "traditional" approaches

Aren't we facing similar risks in "traditional" non-OT developments? Yes, many of the risk factors and the associated mitigation strategies are known from traditional projects. However, in the results of numerous risk assessments conducted on OT projects, a consistent pattern emerged. Figure 1 shows the process followed. SRE-s were conducted, and the results were structured around impact areas

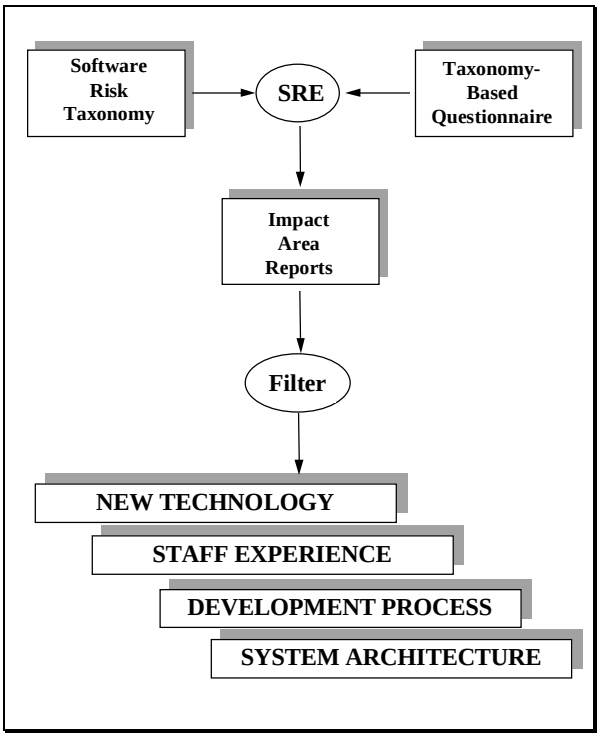


Figure 1:
Process used to determine the "vital few" OT risk factors

specified by project management (See APPENDIX.) We would like to emphasize that the material presented in the Appendix is taken from actual projects. We made only minor changes to assure the required non-attribution, combined findings from different projects and eliminated some redundancies, but did not make up any issues. All observations and their risk magnitude are shown in the same format as they were documented on the flip-charts during the SRE sessions.

---

[4] P. Hantos, Experiences with the SEI risk evaluation method, Proceedings of the '96 Pacific Northwest Software Quality Conference, October 1996, page 69-79.

Studying the impact area reports and consolidating the issues according to their root causes we identified the following "vital few" factors for large projects adopting OT:

- The selection and deployment of any new aspects of object technology should be carried out within a planned, disciplined, and controlled environment.
- A sufficient number of experienced people are required among managers, architects, and engineers for the overall effectiveness of the development team.
- The development process should be defined and tailored at the outset to suit the experience level of the staff and the capability of the tools in the development environment.
- The system architecture should be sufficiently evolved and stable before proceeding with full-scale implementation.

NEW TECHNOLOGY

Most projects will face critical decisions in new technology -- some of which are new to the project and others which are new to the industry. In our risk evaluations we have encountered two types of risks: (i) the selection of technology which is not compatible with the project's experience-base and development environment new to the industry and not mature enough for large projects, and (ii) overloading the project's staff by introducing several new technologies simultaneously. Our experience indicates these risks can be effectively managed by selecting and deploying new technology for the project in a planned and controlled environment, bearing in mind these findings:

- Technological transition from an existing technology base to OT requires a long-term perspective and strategic planning.
- It is important to gain practical experience with new technology on a small-scale or experimental basis before adopting it on large projects.
- Customer acceptance of OT-based innovative products has to be balanced against the real or perceived business need for being first-to-market.

Technology selection

Projects adopting OT for the first time will have to select from a variety of methods such as Booch, Jacobson, or Rumbaugh, and programming languages or environments such as Smalltalk, Eiffel or C++. Although many of these have been successfully used in building large systems by industry experts, projects may have to select the technology most suitable for their purposes. Projects should consider various factors to enable their transition, including the experience and capacity of their staff and capability of the tools in their development environments.

In addition, projects may have to select technology new to the industry such as distributed objects and standards, i.e., Object Management group's Common Object Request Broker Architecture (CORBA) and Microsoft Corporation's Common Object Model (COM.) Projects may also have to select from new products and tools of third-party vendors, e.g., CORBA implementations such as NEO from Sun Microsystems and ORBIX from Iona Technologies. There are many uncertainties and risks. Will new products from third-party vendors be robust enough and their defects fixed in time or will the project be at the mercy of these vendors? Will the tools integrate with the project's existing development environment or will they require costly workarounds? Will CORBA or COM be accepted by the market

as the standard object request broker? It is very unlikely that the market will support more than one distributed object standard in the long run [5]. Strategies to mitigate technology selection risks include:

- Pilot any new technology on a small scale before adopting them on large projects. For example introduce new technology within ongoing projects as a controlled experiment with specified variables so future projects can benefit from the experience. OT is special in the sense that many times it is perceived as a silver bullet to all ills of software development, and as a result the temptation is too serious to introduce it on a full scale.
- Use tools such as S-curve analysis to decide the appropriateness of a given technology. The concept of S-curves was first introduced more than a quarter century ago to plot the progress of a new technology as a function of time [6].

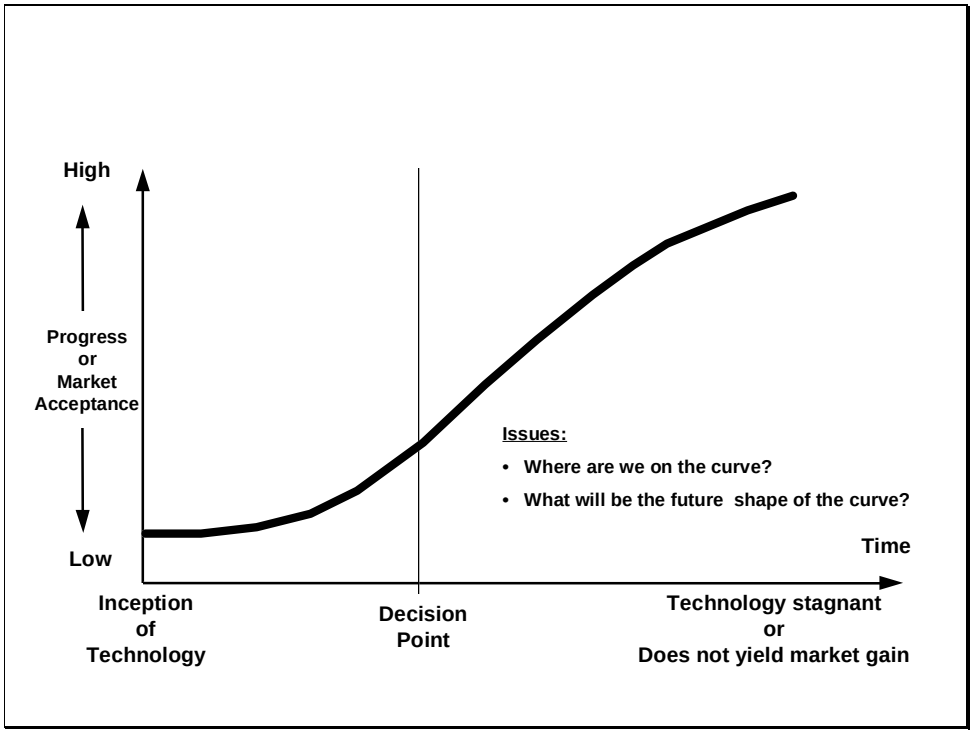


Figure 2:
Using technology S-curves

The plot in Figure 2 shows that during early stages of a new technology, progress is slow, but as a critical mass of engineering expertise builds up, progress can be rapid, even exponential. Using this tool, multiple S-curves for related technology can be plotted on the same axes; gaps in the different S-curves may indicate relative technology benefits, guiding the decision-maker to select the appropriate technology.

---

[5] A. Helal and R. Badrachalam, COM versus CORBA: will Microsoft come out on top?, IEEE Computer, October 1995, pages 61-62.

[6] P. Asthana, Jumping the technology S-curve, IEEE Spectrum, June 1995, pages 49-54.

Technology overload

A problem that we have seen even among projects otherwise prudent about selecting appropriate technology, is one of simultaneously adopting several new technologies. This is common to projects that are attempting to make a technological leap to gain competitive advantage in the marketplace. Each new technology by itself may have a manageable set of risks but the combined effect of individual risks from several technologies dramatically increases the overall complexity of the project. In such situations, there are too many variables that are collectively difficult, if not impossible to control. The project's resources are also stretched to the limit, leaving no buffers to deal with the increased complexity and risks that surface late in the project development.

Projects should take a balanced approach when adopting multiple new technologies. The following may be useful to mitigate risks:

- Define incremental technology goals such that subsequent projects can build on the experience gained from its predecessors. For example, initial projects can adopt foundation-level technology such as object-oriented analysis and design methods and defer new technologies such as distributed objects to later projects.

- Focus on adopting technology to gain customer acceptance instead of being first-to-market. According to Asthana [6], the acceptance of innovative products in the market appears to have a pattern similar to the technology S-curve shown earlier. During initial stages market acceptance is slow because only a small number of customers are willing to take the risk and the majority may accept it only over a longer period of time. Projects can use this information to their advantage by staggering their OT adoption, focusing initially on that part of the technology which will help in gaining customer acceptance and deferring other technologies to subsequent projects.

STAFF EXPERIENCE

For large projects to adopt OT effectively, they should ensure availability of sufficient number of managers, architects, and engineers who are proficient in the new technology. Large projects are often staffed by people who have worked many years in the effected application domain, but do not have enough OT experience. Some highlights of the lessons learned are as follows:

- Adopting a new paradigm takes far more time than what projects typically plan for.

- Development teams are more effective when the proficiency of individuals is on the same level

- People gain experience when they are allowed to make mistakes and large projects typically do not provide such an environment.

Paradigm shift

Adopting OT requires a change in mindset, particularly when managers, architects, and engineers are experienced in other methods such as structured analysis, design and programming methods. For example, the concept of decomposing problems into roles and developing objects that encapsulate those roles is radically different from the concept of solving problems via functional procedures. We found that

large programming environments are typically not suited for learning new paradigms, because of the high pressure to become productive quickly. Although training can impart some of the required skills, when faced with schedule pressure, most people tend to revert to methods that are more familiar.
To foster learning a new paradigm, these strategies may be employed:

- Provide foundation-level training up front before scheduling project-specific training. Although usually the project's OT strategy commits to a particular programming language and OOA/D methodology, experience shows that a solid, and methodology-independent foundation is critical, and most training vendors will be happy to cooperate and customize their programs for on-site offerings. People who attended foundation classes before they were exposed to the nitty-gritty's of actual methodologies and tools, were much more efficient in their work, had a much better understanding of the concepts, and were able to view and evaluate methods without being locked into the details of particular notations. They were also in a much better position to change methodology or tools when needed. For example Jim Odell and Associates put together such a program for Xerox, based on Jim Odell's book [7]. The course was offered in two versions: short version, to provide introduction and raise awareness, and a longer one, to analyze and evaluate state-of-the-art methodologies in greater detail. His seminars provided an advanced, comprehensive guide to exploiting object orientation, including independent evaluations of the latest directions and new products, including CASE tools, O-O databases, and emerging O-O standards.

- Create an environment for people to learn incrementally and through mentoring programs. For example, on the OS/400 project at IBM, classroom training was augmented with mentoring activities and almost half of the total training and education time was allocated for mentoring activities [8].

Experience base

People have different learning curves and do not all reach the required level of proficiency at the same time. We find projects adopting OT usually do not factor variable learning curves in their plans and typically allocate fixed training time. The assumption that teams become productive after their training period may not be true in practice as it takes much longer for people to function together, particularly when individual skills are not at parity with each other. We also find that some skilled personnel may become less productive as they become overburdened with work or being reassigned to teams short of skills. Ultimately, developers and managers must gain hands-on experience in order to become productive. This is obtained largely by working in an environment where there is room to make mistakes and learn the process. On large projects, people are afraid to make mistakes because of the project's visibility and the impact of errors. The result is that they take more time to perform their work or they are forced to make costly mistakes, either of which can be avoided if they had adequate experience.

We believe large projects should not be used to forge staff experience because of the high overhead costs, instead, we recommend the following strategies:

- Allow individuals to gain experience on small scale or experimental projects before moving to larger projects. This will enable people to gain critical experience in an environment that allows them to make mistakes and to learn from them. It will also avoid costly rework and delays on large projects.

---

[7] J. Martin and J. Odell, Object-Oriented Methods: A Foundation, Prentice Hall, 1995.

[8] W. Berg, M. Cline, and M. Girou, Lessons Learned from the OS/400 OO project, Communications of the ACM, October 1995, Vol. 38, No 10, pages 54-64.

- In areas where critical skills are lacking, augment the project's base-level experience with outside experts. Identify "skills gaps" early in the project and recruit experts or hire consultants to perform the work.

- Plan for "Pilot to Production" (Organizational scaling) and use small or experimental projects as training ground for developers and managers. This offers an opportunity for people to gain sufficient experience before being placed on larger projects and also provides an environment that is more tolerant of mistakes.

- Plan to train personnel explicitly on how to deal with legacy issues and reconcile the differences between the old and new methods and tools. Also provide specific training on the compliance to standards if needed, for example on how to migrate to ANSII or POSIX.

- Plan for phased-in O-O language introduction (Language scaling) to satisfy the Just-In-Time training delivery requirements. This is particularly important and useful in case of C++ or other complex languages

- Classify the training audience and plan for customized training. Beside some self-explanatory groups, like software architects, managers and developers, we found that successful projects created interdisciplinary training groups for marketing, product planning, and systems engineering personnel to introduce the new paradigm. O-O projects require a sound and well designed training program. While the various training-blocks can be organized and offered on corporate-level by centralized training organizations, the actual training strategy and the customization of the building blocks require a high level of understanding of the product details and the organizational dynamics. This structure also allows us to better comprehend and deal with the issues of legacy systems, programming languages and tools. Last but not least, there is no substitute for a well trained and loyal development team.

DEVELOPMENT PROCESS

The importance of a well-defined process, particularly in large projects, is documented in the work of many OT experts [9, 10]. Some projects tend to have risks because of how they use the iterative incremental approach or because their technical and management processes are not well defined.

We find large projects adopting OT are lacking in both process stability and project control. To facilitate stability and control of the project, the following process enablers should be adopted:

- To avoid costly rework and project delays, the architecture should be at a fairly mature and stable version before proceeding to implement it on a production scale.

- Adopting basic software management practices in the OT environment helps to maintain project control.

---

[9] I. Jacobson, Object-oriented software engineering: a use case driven approach, ACM Press and Addison-Wesley Publishing Company, Reading, MA, 1992.

[10] G. Booch, Object-Oriented Analysis and Design with Applications, The Benjamin-Cummings Publishing Company, Inc. 1994.

Process stability

Large projects adopting OT require stable development processes to effectively and efficiently carry out the development work. Some projects have unstable processes because of how they use the iterative incremental approach to simultaneously develop the architecture and implement it on a production scale. For example, most OT methods require a certain degree of architectural refinement, during which a few developers implement the architecture to provide feedback to the architects (or they might do it themselves ...) Some projects staff to production level prematurely. The high overhead costs associated with the excessive staffing level create pressure to begin implementing the architecture before it is sufficiently evolved and stable. This has resulted in large amounts of throw-away code due to frequent architectural changes, budget overruns and project delays.

Large projects adopting OT for the first time tend to define their development process as they go along, particularly because they discover what does and does not work for them only after they start implementation. However, introducing any process changes midstream creates a host of difficulties which includes the communication and coordination of work assigned to the development teams. We recommend the following strategies to ensure process stability:

- At the outset, ensure that experienced process specialists define and tailor the development process to suit the skills and experience of the staff and the capability of the tools in the development environment.

- Evolve the architecture to a fairly stable version and validate it before proceeding to implement the system on a production scale.

Project control

Sound software management practices apply equally to OO as well as non OO systems, including basic practices such as SCM, SQA and peer reviews [10]. But is it true that maintaining control in OT projects is more difficult than in traditional projects? The answer is yes. OT most likely requires more formal approaches than any previous methodology. One of the critical objectives of most mainstream approaches is to give many different views of the system, and via systematic steps ensure consistency and synchronization among those views. This aspect of OT results in a high-complexity, large number of design artifacts. We find there is some inadequacy even in the implementation of basic practices within OT projects and some managers have difficulty maintaining project control. For example, in one project there were no acceptable checklists or guidelines available to conduct peer reviews on work products such as object design documents and object implementations.

The recommended mitigation strategies are as follows:

- Establish a framework to manage and maintain control of the project. This is facilitated by translating high-level project goals to quantifiable implementation-level goals for developers; ensuring traceability of product features using mechanisms such as use-case scenarios to document the expected behaviors in architecture components, classes, and object diagrams [9]; and providing project-specific guidelines and checklists for developers regarding common implementation patterns.

- Implement a practice of continuous risk management across the project including identifying risks as they become known to individuals, prioritizing issues, assigning responsibilities, planning corrective actions, and tracking items to closure. For example, the SEI promotes a paradigm to identify,

analyze, plan, track, control, and communicate risks. Figure 3 on the next page illustrates this paradigm, in which the activities are shown as a cycle, indicating a continuous operation throughout the project's duration. Note that communication is at the hub of this paradigm because of its primary importance in performing and integrating other activities.

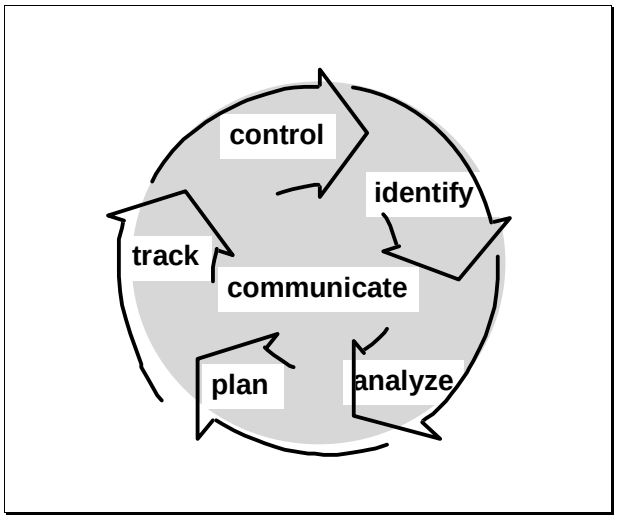


Figure 3:
SEI Risk Management Paradigm

Planning for verification

Many good resources are available to review the details of various technical problems associated with testing object oriented systems, for example Binder [11] and Chen at al. [12]. During the SREs though we learned, that there were some seemingly simple, meta-level process issues, which had to be recognized and resolved:

- SQA function was undefined or not well defined for the project
- Object Oriented SQA expertise is rare, and the specialty is not acknowledged
- Implementation starts, but no formal acceptance criteria is defined for the objects
- Test plans are not developed, or their development starts too late
- It is assumed that O-O testing will be just like the "old stuff"
- No change control for feature specs; lack of traceability will hinder testing
- Lack of adequate component test tools

---

[11] R. V. Binder, Trends in testing object-oriented software, IEEE Computer, Ocotber 1995, pages 68-69.

[12] Chen, Chris; Kim, Young-Si; and Song, Young-Kee, developing an Object-Oriented Software Testing and Maintenance Environment, Communications of the ACM, October 1995, Vol 38, No 10, pages 75 - 86.

SYSTEM ARCHITECTURE

Large projects adopting OT are particularly at risk at design, because there are no “right’ ways to craft the system’s architecture. Experience indicates current OT methods are useful in capturing good designs, provided the architect can come up with such a design [13]. Our own experience reinforces the importance of having highly skilled and experienced architects on the project to ensure system integrity and performance.

Design integrity

Achieving project goals, including long term goals such as maintainability and extensibility are dependent on how critical system features are designed. Good design includes making trade-off decisions between conflicting requirements and between potential system features as well. We find some projects postpone such decisions either because their architects are inexperienced or because of time pressure. For example, in one of the projects evaluated, important system features such as security, crash recovery, and diagnostics were postponed to a later release to accommodate near-term delivery schedules. Another type of risk arises from trying to build large systems from scratch by including system capabilities to meet all of the user requirements at once. In such situations, projects may not be able to control the complexity of the architecture, making it difficult for developers to understand, build, test, and maintain the system.

The completeness of the architecture has to be validated. In our opinion it is essential to subject the proposed architecture to a systematic peer-review process. Again, we can not emphasize enough some unique benefits of the SRE method. The people who are inside the organization are the best sources of identifying the problem areas of the design. The development team members have to both understand and accept the architecture, and the peer-review process facilitates both.

Ivar Jacobson promotes a special, use case driven O-O design approach [9]. While Jacobson's goal was to provide a complete software engineering process framework, our risk mitigation teams found relevant ideas in his work for projects where the use case design approach itself was not applied. Appropriately designed use case scenarios, tracking and tracing the system object life-cycles were successfully applied to validate software architecture.

Another delicate question is to what extent should simulation be relied upon. Many feel that in the case of large, complex systems, integrated with large chunks of third party code, sometimes it is more cost effective to proceed with an iterative design process. In the case of such process -- promoted, for example, by the Rational Rose tool set and its embraced O-O methodologies [10] -- we can move freely between analysis, design and coding, and experiment with the actual code instead of a simulated model. Proponents of the Shlaer-Mellor method [14] on the other hand praise the idea of translation-based, concurrent design, where developing a somewhat complete application model is a must, and a wide variety of tools will facilitate the dynamic verification of such models via interactive simulation.

---

[13] C. M. Pancake, The promise and the cost of object technology: A five-year forecast, Communications of the ACM, October 1995, Vol. 38, No 10, pages 33 - 49.

[14] S. Shlaer and S. J. Mellor, The Shlaer-Mellor Method, Technical Report, Project Technology, Inc., 1996.

Mitigation strategies to ensure design integrity include:

- Start small, and practice reuse. Use objects from simpler systems that worked previously as the primitive building blocks for larger and more complex systems.

- Validate the architecture using a standard review practice before proceeding to implementation on production scale. For example, the former AT&T Bell Laboratories used to advocate a systematic and disciplined practice to formally review and evaluate the architecture in two stages: first, after the high-level design of the system is completed; and second, after implementation is underway and some initial performance measurements are available from the system [15].

- Use a scenario-based approach for architecture validation to augment the inspection process.

- A combination of SRE and simulation are recommended to augment the validation process.

System performance

In most projects, system performance is a critical requirement. For example, the number of pages printed per minute is a key metric for projects developing printing or reprographic systems at Xerox. Projects adopting OT may have to contend with many system performance factors such as design constraints, new hardware and software, and middleware from third-party vendors. System performance in object-oriented systems is challenging even among expert architects and developers. Industry experience indicates many performance risks including overhead in sending messages between objects, multiple layers to invoke methods, inheritance lattices, paging and cache behavior during run time, and the dynamic allocation and destruction of objects [10]. Our own experience also indicates performance risks due to message passing and inheritance complexities.

Projects building distributed object systems may face additional performance risks due to the relative immaturity of the concepts, unknown middleware-behavior and design issues of distributed objects. For example, in one project, the granularity of distributed objects was identified as a cause of performance problems. Projects have very few resources and benchmarks available for planning and assessing performance of such systems. CORBA implementations are relatively new and though some performance measures have been taken, there are many untested areas with regard to message size, object load, maximum number of servers, etc.

In O-O projects with identified performance problems, there seems to be little of what is described as essential performance engineering. There appears to be a lack of trade-off analysis of the plug-in extensibility versus the necessary performance criteria. In short, an up-front performance modeling is missing as a definition of component level performance criteria and performance prototyping. Also, overloaded development teams might defer the solution of performance issues to the hardware vendor, or new releases of the operating system, introducing further risk factors.

---

[15] Best Current Practices: Software Architecture Validation, AT&T Bell Labs internal BCP Handbook, copyright 1993, AT&T.

Based on our experience the following mitigation strategies proved to be useful in dealing with performance-related problems:

- There is an executed and well managed software (and system) performance engineering plan in place that stretches across the entire development life cycle and includes modeling, measurement and tracking.

- Performance reviews are carried out by knowledgeable technologists and domain experts.

- External dependencies are carefully monitored and alternative plans are developed in advance.

CONCLUSION

Large projects adopting OT are vulnerable to certain risks, but most of them can be actively managed and mitigated. The more common risks are found in the areas of new technology, staff experience, development process and system architecture. We have discussed some of the risks and suggested a few mitigation strategies. Large projects adopting OT can increase their likelihood of success by ensuring that: (i) the selection and deployment of a new technology is planned and controlled; (ii) there is sufficient number of experienced managers, architects, and engineers on project teams; (iii) the development process is defined and tailored at the outset to suit the experience level of the staff and capability of the tools in the development environment; and, (iv) the system architecture is sufficiently evolved and stable before proceeding to implement it on a production scale.

ACKNOWLEDGMENTS

This paper is based on the past efforts of many people who have participated with us on software risk evaluations. It would be remiss on our part if we did not acknowledge their contributions, particularly those team members whose ideas were used to develop the presented mitigation strategies.

We thank Bob Factor, Maurice Holmes and Charlie Sie from Xerox and Ron Higuerra from the SEI for encouraging and supporting our efforts. We also thank Rebecca Bower, Marvin Carr, Dave Gluch, Bob Lang and Bill Wood for their comments that helped us to improve the quality and readability of this paper.

APPENDIX

Sample Impact Area Report

Reliability & Quality Impact Area

Issues and Concerns from the SRE sessions	Risk Magnitude
No formal review and assessment of the architecture against performance, reusability, functionality, etc.	7.5
Insufficient time to climb the learning curve. Schedule pressures cause inefficiencies	7.0
There is no consistent process for generating and approving requirements	6.8
No change control on feature specifications	6.8
Lack of well-defined requirements prevents us from developing a solid product	6.5
Integration/testing process is not mature and capable to handle all the problems, such as large number of objects, complexity, concurrency, scaling, security, recovery, etc.	6.5
Developers do not have a big picture or high level view of development process	6.4
The detail in the requirements (features specification) was not developed quickly enough - the incoming new requirements are further complicating the situation	6.3
Allocation of the required resources to develop 100% of component test tools has not been allocated -- risk to testing schedules, quality and overall program schedule	6.3
Architecture is incomplete	6.1
No formal SQA process defined - code reviews have not been done correctly, no acceptance criteria	6.0
No SQA, formal reviews, audits, appropriate code standards to follow at every area	5.8
Code and development process are developed concurrently	5.8
Not all of the feature specifications are complete and under change control	5.5
Unproved test strategy, too complex test environment, lack of manpower	5.5
Lack of OO SQA expertise	5.4
Lack of application & OO domain knowledge to review architecture on system level	5.3
Coding starts without obtaining sign-off on feature specifications	5.3
Requirements baseline is missing and not under change-control	5.3
No formal, released test environment and support	5.3
Totality of the feature specs was not reviewed to ensure coherence and completeness	5.1
Mapping generic requirements into more detailed feature specifications is not happening as fast and completely as it would be desirable and does not generate the required traceability matrix (connection between objects and features).	5.0
System is so tightly-coupled, that it makes it difficult to prototype - in some cases there is not enough time for prototyping	5.0
Changes in development environment, hardware, external interface communications	4.7
Standard design notation is being modified in house -- source of confusion	4.5
Integration processes not fully defined	4.3
Many of the required tools are either not compatible and actually non-existent	4.1
No adequate document control system - an extension library is needed on top of SCM	4.1
Lack of buy-in to a unit test strategy. Inadequate testing will cause integration problems	4.0
Lack of sufficient integration & test plan history, thus previous experience is lost	3.5

Maintainability & Extensibility Impact Area

Issues and Concerns from the SRE sessions	Risk Magnitude
Crash recovery, accounting/billing, system startup, security is not architected	7.9
There is a tendency to trade-off long-term goals for short-term needs	7.6
Architecture review process is broken	6.8
Extremely high risk of producing a “core” architecture for a fleet of products.	6.8
Architecture is too complex for maintainability	6.8
The architecture is not providing complete formal requirements to the development teams e.g. performance, system external interfaces, maintainability, extensibility, quality	6.7
Complexity of architecture (or representation, degree of detail) and how it affects implementation -- people’s perception affects time of implementation, time for mentoring.	6.4
Engineering organizations are uncomfortable with the architecture -- technical complexity, representation of the architecture, performance, etc.	6.3
Communication to developers of long term vision is missing	6.3
Maintainability issues haven’t been considered	6.1
Architecture documentation is not complete: rationale is not documented	6.1
There is a lack of experience and skills in system engineering	6.0
Developers do not have sufficient knowledge to review and challenge the architecture	5.9
Conflicting requirements cause the architecture to be complex	5.7
Postponement of feature decision impacts extensibility	5.6
Content and implementation of the architecture is too complex	4.6
Problem with development model - emphasis on tactical issues, “blurred” big picture	4.5
Features in most cases are unfamiliar	4.1
No clear definition of architecture activities, no commitments or schedule from architecture team -- architecture plan is missing	4.0
The short-term view when making technical decisions affects maintainability requirements such as remote debugging.	4.0
If critical feature specifications are not done by the end of the year then there is potential for the architecture to change	3.9
Communication of product vision to senior management needs to be reinforced	3.1
The techniques for maintaining C++ code may not be understood	3.1
Maintenance business processes may not be ready in time to support	3.0
Concern about open systems requirement, specifically controlling security	2.3

Cost & Schedule Impact Area

Issues and Concerns from the SRE sessions	Risk Magnitude
For large parts of the organization lack of experience in the new technology, methodology, hardware and tools	7.7
Current plan is schedule driven and bottom-level plans (business level plan vs. engineering level plan) does not support it and management is not prepared to reconcile	7.5
Target schedule of top-level plan is unrealistic and not based on experience to-date.	7.5
Inability to estimate effort required (lack of experience with estimation of OO projects) causes concerns for schedule -- coupled with the issues around using the evolutionary architecture	7.1
Extensibility (flexibility), distribution, plug-n-play, performance can be incompatible requirements and requires trade-offs among these -- causes a lot of rework	7.0
Not enough people on the architecture team	6.8
Feedback loops from implementer to architecture team is too long	6.3
So far, iterations do not build on each other, very little “reuse” from one to the next	6.3
Capacity of the development system (network bandwidth, speed of compilation) is a concern -- impacts schedule	6.1
Developers are expected to learn a whole new environment -- requires time, effort, & developer buy-in	5.9
Current architectural development cycle is unpredictable and becoming a bottleneck	5.9
Schedule driven project management coupled with lack of prioritization	5.8
SCM is not sufficiently staffed -- unknown learning curve	5.8
SCM is not ready to support the ongoing activities	5.6
Violation of the first rule of OOT introduction: “Start Small.”	5.6
Developers lack of confidence in the current plans	5.6
Object Implementation Package requirements are being derived by the Object Implementation Team and the process requires many reviews and long analysis cycle	5.5
Development environment is not stable, some tools are not integrated and operational. could impact schedule	5.4
Spiral model is not being used properly -- architecture has been thrown away each time	5.3
Reassignment causes loss of knowledge and new learning curve	5.1
Lack of parallelism in some of the subgroups of the program	5.0
Architecture milestones do not meet schedules.	5.0
Near-term focus of decisions impact downstream preparation for integration	5.0
Software tools and infrastructure is not in place (skills, budget and people)	5.0
Performance of software tools and development environment is below expectations. (‘slow’, network traffic issues, stability, tool crashes, distributed debugging capabilities)	5.0
Concern regarding testing with respect to effort required	4.8
Processing delays in equipment and tools acquisition, also overall cost is underestimated	4.6
There are too many external vendor dependencies	4.6
Lack of availability of adequate number of software licenses	4.3
Lack of processing power in the development workstation and not enough PCs	4.3
Some development teams do not have access to the SCM, problem tracking tools -- possible delays, loss of time, impact schedule	3.9
The prototype approach to develop architecture has caused unnecessary rework	3.9

Performance Impact Area

Issues and Concerns from the SRE sessions	Risk Magnitude
Performance is a concern. The areas include architecture model, performance tuning of OS and other areas, decomposition, special output drivers, etc.	8.6
All of the new technologies together are untested and may affect performance.	8.1
Unless something changes, and management makes some decisions regarding what to keep, especially distribution/plug-n-play/concurrency, the performance requirements cannot be met	8.0
No component level performance criteria	7.5
Distributed requirements (resources access and component plug-n-play) are driving to use system in such a way as to degrade performance	7.0
There is no small scale prototyping at the architectural level -- e.g. commonly used data structures and methods, and in critical performance areas	6.2
Lack of integrated, extensive performance analysis	5.5
Performance modeling tools are non-existent for this environment	5.0
Using distributed objects has a negative impact on portability and performance	5.0
The correct hardware processor options may not be available in time to validate with software and this is required to achieve performance	4.0
Some requirements defined in the architecture are technically difficult to implement effectively	3.8

Other Impact Areas

Issues and Concerns from the SRE sessions	Risk Magnitude
Developers perceive that the required fundamental changes will not be made	8.6
Too many areas & not many of them are mature enough (architecture, OT, C++, SQA)	7.9
No project level goals for reliability / maintainability / extensibility / performance	7.9
Organization does not perceive themselves as being a team -- lack of accountability	7.5
Morale is low on program -- causes are personality of architecture team; ineffective utilization of resources; lack of orchestrated management; lack of accountability	7.5
Roles and responsibilities between business and engineering team are not bought-in	7.4
New environment where there is no process experience requires a new process -- and aspects of development are being defined concurrently with product development	7.2
Lack of OO specific technical and management skills	7.1
Inputs & outputs and roles & responsibilities for the groups are not well defined	7.0
Lack of maturity of technology and the capability of the organization	6.3
The few technically qualified people are being overused	6.2
Morale issues: People resent the schedule driven environment	6.1
Perception that the current OO approach will not achieve program's quality, cost and delivery goals, and this impacts morale	5.9
Separation of architecture and development teams causes lack of interaction	5.9
Dependency on a few key people	5.8
Some development team members perceive architecture is forced upon them and this causes resentment, lack of cooperation	5.6
Lack of experience & involvement of project management from the top and second-level management (on managing similar kinds of projects using OT)	5.2
Lacking of technical skills in OO testing impacts schedule and quality	5.1
Key people being overworked impacts burnout, morale	5.0
A lot of fingerpointing, lack of communication between architecture team and developers	5.0
Good news: People are bringing up issues with the architecture - Problem: No process in place to resolve those issues	4.9
Entrepreneurial expectations, but the environment, incentives are not entrepreneurial	4.9
Developers do not have the comprehension of the complete system	4.8
No reuse strategy and process in place within the current program and future	4.5
Object mentality and competence has not reached the necessary level	4.5
Lack of authority (if not directed by management) to resolve problems	4.4
Evaluation of offshore team's effectiveness is not happening and causes morale issues	4.2
Problem tracking system is too new for the people	4.1
Organizational instability in the larger context could impact the project's goals	4.0
Senior programmers do not have sufficient time to provide more help with the complexity	3.5
Using architecture as the scapegoat for other project shortcomings	3.4
The need for supporting tools in a geographically diverse organization increases the operational complexity (Process and security issues for the SCM tool)	3.3
Concerned about schedule -- when we get to market it will not be the sizzle we expected	3.1

Software Metrics: Ten Traps to Avoid

View Slides

Karl E. Wiegers, Ph.D.

Eastman Kodak Company
901 Elmgrove Road
Rochester, NY 14653-5811
Phone: (716) 726-0979
Internet: kwiegers@kodak.com

ABSTRACT

Implementing a software metrics program is a challenge. Both the technical and the human aspects of software measurement can be difficult to manage. This paper examines ten traps that can sabotage the unsuspecting metrics practitioner. Several symptoms of each trap are described, along with several suggested strategies for preventing and dealing with the trap.

The traps discussed are: lack of management commitment; measuring too much too soon, or too little too late; measuring the wrong things; imprecise metrics definitions; using metrics to evaluate individuals; using metrics to motivate, rather than to understand; collecting data that is not used; lack of communication and training; and misinterpreting metrics data. By staying alert to these common risks, you can chart a course toward successful measurement of your software development activities.

BIOGRAPHY

Karl E. Wiegers is a software process engineer in a large product software division at Eastman Kodak Company in Rochester, New York. His 18-year Kodak career has included positions as a photographic research scientist, software developer, and software manager. Karl has led various software process improvement, quality, and measurement efforts at Kodak since 1990. Karl received a B.S. degree in chemistry from Boise State College, and M.S. and Ph.D. degrees in organic chemistry from the University of Illinois. He is a member of the IEEE Computer Society and the ACM. Karl is the author of the award-winning book *Creating a Software Engineering Culture* (Dorset House, 1996), as well as over 100 articles on many aspects of computing, chemistry, and military history. He is a frequent speaker at software conferences and professional society meetings.

This paper was originally published in *Software Development*, October 1997. It is reprinted (with modifications) with permission from *Software Development* magazine.

Software Metrics: Ten Traps to Avoid

Karl E. Wiegers

As software development gradually evolves from art toward engineering, more and more developers appreciate the importance of measuring the work they do. While software metrics can help you understand and improve your work, implementing a metrics program is a challenge. Both the technical and the human aspects of software measurement are difficult to manage.

According to metrics guru Howard Rubin, up to 80 percent of software metrics initiatives fail [Rubin, 1991]. This article identifies ten traps that can sabotage the unsuspecting metrics practitioner. Several symptoms of each trap are described, along with some possible solutions. By being aware of these common risks, you can chart a course toward successful measurement of your organization's software development activities. However, it is important to realize that none of the solutions presented here are likely to be helpful if you are dealing with unreasonable people.

Objectives of Software Measurement

As you design, implement, and adjust your software metrics program (and it *will* need adjusting as time goes on), it's important to keep these four primary objectives of software measurement in sight:

1. To collect objective information about the current state of a software product, project, or process.
2. To allow managers and practitioners to make timely, data-driven decisions.
3. To track your organization's progress toward its improvement goals.
4. To assess the impact of process changes.

It is easy to lose sight of these objectives and collect data just to say you have it, or just because you are able to measure it. Emphasize the business and technical results you are trying to achieve, link the metrics program to your software process improvement activities, and make sure the data you collect gets used for constructive purposes.

Trap #1: Lack of Management Commitment

Symptoms: As with most improvement initiatives, management commitment is essential for a metrics effort to succeed. The most obvious symptom that commitment is lacking is when your management actively opposes measurement. More frequently, management claims to support measurement, and effort is devoted to designing a program, but practitioners do not collect data because management hasn't explicitly required it of them.

Another clue that managers aren't fully committed is that they charter a metrics program and planning team, but then do not assist with deploying the program into practice. Managers who are not committed to software measurement will not use the

available data to help them do a better job, and they won't share the data trends with the developers in the group.

How can you distinguish true commitment from lip service? First, look for allocation of resources, including capable people (not just whoever happens to be free at the moment) and money for tools. Necessary tools include metrics analysis software for source code, a database to store the collected data, charting and reporting software, and scripts to automate the data collection and analysis processes. A committed manager will also issue a policy to the organization, clearly stating the objectives of the metrics program, emphasizing his personal interest in the program, and stating his expectations of participation. A committed manager will help the program succeed by overcoming the resistance that mid-managers and project leaders may exhibit to the measurement initiative. This is virtually impossible to accomplish from the bottom up, so the drive to succeed must come from senior management.

Solutions: To persuade managers of the value of software measurement, educate them! Good resources to start with are *Software Metrics: Establishing a Company-Wide Program Practical Software Metrics for Project Management* [Grady, 1987] *and Process Improvement* [Grady, 1992]. Tie the metrics program to the managers' business goals, so they see that good data provides the only way to tell if the organization is becoming more effective. You also need management's input to help design the metrics program, to ensure that it will meet their needs. Those selecting the data items to collect should attempt to hear the "voice of the customer" from the managers at various levels who constitute primary audiences for the program. Ask the managers to sketch out the kinds of charts that would help them better understand the software activities in their organization. The inputs required to generate the charts will guide the selection of data items to be measured.

If you cannot obtain commitment from senior management, turn your attention to the project and individual practitioner level. There are many valuable metrics that developers and project teams can use to understand and improve their work, so focus your energy at those who are willing to try. As with any improvement initiative, grass roots efforts can be effective locally, and you can use positive results to encourage a broader level of awareness and commitment. However, expect to run into a glass ceiling that will limit the potential organizational breadth of the metrics program unless managers at various levels help make it happen.

Trap #2: Measuring Too Much, Too Soon

Symptoms: Hundreds of aspects of software products, projects, and processes can be measured. It is easy to select too many different data items to be collected when beginning a metrics program. You may not be able to properly analyze the data as fast as it pours, so the excess data is simply wasted. Those who receive your summary charts and reports may be overwhelmed by the volume and tune out.

Until a measurement mindset is established in the organization, expect resistance both to the concept of measuring software, and to the time required to collect, report, and interpret the data. Developers who are new to software measurement may not believe that you really need all the data items you are requesting. A long list of metrics can scare off some of the managers and practitioners whose participation you need for the program to succeed.

Solutions: Begin growing your measurement culture by selecting a fairly small, balanced set of software metrics. By balanced, I mean that you are measuring several complementary aspects of your work, such as quality, complexity, and schedule. As your team members learn what the metrics program is all about, and how the data will (and will not) be used, you can gradually expand the suite of metrics being collected. Start simple and build on your successes.

The participants in the metrics program must understand why the requested metrics are valuable before they'll want to do their part. For each metric you propose, ask, "What can we do differently if we have this data?" If you can't come up with an answer, perhaps you don't need to measure that particular item right now. Once the participants are in the habit of using the data they collect to help them understand their work and make better decisions, you can expand the program.

Trap #3: Measuring Too Little, Too Late

Symptoms: Some programs start by collecting just one or two data items, which may not provide enough useful information to let people understand what's going on and make better decisions. This can lead stakeholders to conclude that the metrics effort is not worthwhile, so they terminate it prematurely.

Another obstacle to getting adequate and timely data is the resistance many software people exhibit toward metrics. Participants who are more comfortable working undercover may drag their feet on measurement. They may report only a few of the requested data items, report only data points that make them look good, or turn in their data long after it is due. The result is that managers and project leaders don't get the data they need in a timely way.

A metrics program has the potential to do actual damage if too few dimensions of your work are being measured. People are tempted to change their behavior in reaction to what is being measured, which can have unfortunate and unanticipated side effects. For example, if you are measuring productivity but not quality, some people may change their programming style to generate more lines of code and therefore look more productive. I can write code very fast if the quality is not important.

Solutions: As with Trap #2, the balanced set of metrics is essential. Measure several aspects of your product size, work effort, project status, quality of product, or customer satisfaction. You don't need to start with all of these at once, but select a small suite of key measures that will help you understand your group's work better, and begin collecting them right away. Since software metrics are usually a lagging indicator of what is going on, the later you start, the farther off-track your project might stray before you realize it. Avoid choosing metrics that might tempt program participants to optimize one aspect of their performance at the expense of others. Make participation in the metrics program a job expectation.

Trap #4: Measuring the Wrong Things

Symptoms: Do the data items being collected clearly relate to the key success strategies for your business? Are your managers obtaining the timely information they need to do a better job of managing their projects and people? Can you tell from the data whether the process changes you have made are working? If not, it's time to reevaluate your metrics suite. Another symptom of this trap is that inappropriate surrogate measures are being used. One example is attempting to measure actual project work

effort using an accounting system that insists upon 40 labor hours per week per employee. Force-fitting an existing solution to a specialized problem will not necessarily give you the high-quality information you need to really understand how your organization is performing its software work.

Solutions: Select measures that will help you steer your process improvement activities, by showing whether process changes are having the desired effect. For example, if you're taking steps to reduce the backlog of change requests, measure the total number of requests submitted, the number open each week, and the average days each request is open. To evaluate your quality control processes, count the number of defects found in each test and inspection stage, as well as the defects reported by customers.

Make sure you know who the audience is for the metrics data, and make sure the metrics you collect will accurately answer their questions. As you design the program, leverage from what individuals or project teams are already measuring. The goal/question/metric (GQM) paradigm works well for selecting those metrics that will let you answer specific questions associated with organizational or project goals. With GQM, you first determine your improvement or business goals. Next, you identify the questions you'll have to answer to determine if you are making progress toward those goals. Finally, you select the metrics that will give you the necessary data to answer those questions. Several companies, including Hewlett-Packard and Motorola, have successfully applied GQM [Daskalantonakis, 1992].

Trap #5: Imprecise Metrics Definitions

Symptoms: Vague or ambiguous metric definitions allow every practitioner to interpret them differently. One person counts an unnecessary software feature as a defect, while someone else does not. Time spent fixing a bug found by testing is classified as test effort by one person, coding effort by another, and rework by a third. Trends in metrics tracked over time may show erratic behavior because individuals are not measuring, reporting, or charting their results in the same way.

You'll have a clue that your metrics definitions are inadequate if participants are frequently puzzled about what exactly they are being expected to measure. If you keep getting questions like, "Do I count unpaid overtime in the total work effort?" you may be falling into this trap.

Solutions: A complete and consistent set of definitions for the things you are trying to measure is essential if you wish to combine data from several individuals or projects. For example, the definition of a line of code is by no means standard even for a single programming language. Much of the published metrics literature doesn't even define what a line of code meant to the organization whose experience is being described. Standardize on a single tool for collecting metrics based on source code; PC-METRIC from SET Laboratories is a good choice (http://www.molalla.net/~setlabs). Versions are available for both the DOS/Windows and Unix environments, for many programming languages. Automate the measurement process where possible, and use standard calibration files to make sure all participants have configured their tools correctly.

A few metrics, such as function points, do have standard definitions available, which makes it easier to compare data collected by your organization with that of other companies for external benchmarking. However, there are even different kinds of func-

tion points, so make sure everyone involved in the data collection is applying the same definitions. A variety of metrics have been defined for use with object-oriented software, also [Lorenz, 1994].

Those designing the metrics program must create a precise definition for each data item being collected, as well as for other metrics computed from combinations of these data items. This is much harder than you might suspect. Plan to spend considerable time agreeing on definitions, documenting them as clearly as possible, and writing procedures to assist practitioners with collecting the data items easily and accurately. Pilot projects are an effective way to test your metrics definitions and collection procedures with a representative sample of metrics program participants.

Trap #6: Using Metrics Data to Evaluate Individuals

Symptoms: The kiss of death for a software metrics initiative is to use the data as input into an individual's performance appraisal. Using metrics data for either reward or punishment, such as rank-ordering programmers based on their lines of code generated per day, is completely inappropriate. When someone knows that the numbers she reports might be held against her, she'll either stop reporting numbers at all, or only report numbers that make her look good. Fear of the consequences of reporting honest data is a root of many metrics program failures.

Solutions: Management must make it clear that the purpose of the metrics program is to understand how software is being built, to permit informed decisions to be made, and to assess the impact of process changes on the software work. The purpose is NOT to evaluate individual team members. Control the scope of visibility of different kinds of data; if individual names are not attached to the numbers, no individual evaluations can be made. However, it is appropriate to include the *activity* of collecting and using accurate (and expected) data in an individual's performance evaluation.

Certain metrics should be private to the individual; an example is the number of defects found by unit testing or code review. Others should remain private to a project team, such as the percentage of requirements successfully tested and the number of requirements changes. Some metrics should have management visibility beyond the project, including actual versus estimated schedule and budget, and the number of reported and open customer-reported defects. The best results are achieved when individuals use their private metrics data to judge and correct themselves, thereby improving their personal software process.

Trap #7: Using Metrics to Motivate, Rather than to Understand

Symptoms: When managers attempt to use a measurement program as a tool for motivating desired behaviors, they may reward people or projects based on their performance with regard to just one or two metrics. Public tracking charts may be pointed out as showing desirable or undesirable results. This can cause practitioners who are using the charts to understand what's up with their software to hide their data, thereby avoiding the risk of public management scrutiny. Managers may focus on getting "the numbers" where they want them to be, instead of really hearing what the data is telling them. As with Trap #3, the behavioral changes stimulated by motivational measurement may not be the ones you really want.

Solutions: Metrics data is intrinsically neither virtuous nor evil, simply informative. Using metrics to motivate rather than to learn has the potential of leading to dys-

functional behavior, in which the results obtained are not consistent with the goals intended by the motivator [Austin, 1996]. Metrics dysfunction can include inappropriately optimizing one software dimension at the expense of others, or reporting fabricated data to tell managers what they want to hear.

Stress to the participants that we must have accurate data if we are to understand the current reality and take appropriate actions. Use the data to understand discrepancies between your quality and productivity goals and the current reality, so you can improve your processes accordingly. Use the process improvement program as the tool to drive desired behaviors, and use the metrics program to see if you are getting the results you want. If you still choose to use measurement to help motivate desired behaviors, be *very* careful.

Trap #8: Collecting Data That Is Not Used

Symptoms: The members of your organization may diligently collect the data and report it as requested, yet they never see evidence that the data is being used for anything. People may grumble that they spend precious time measuring their work, but they don't have any idea why this is necessary. The required data is submitted to some collection center, which stores them it a write-only database. Your management doesn't seem to care whether data is reported or not (related to Trap #1).

Solutions: Software engineering should be a data-driven profession, but it cannot be if the available data disappears from sight. Project leaders and upper management need to close the loop and share results with the team. They need to relate the benefits of having the data available, and describe how the information helped managers make good decisions and take appropriate actions. Selected public metrics trends must be made visible to all stakeholders, so they can share in the successes and choose how to address the shortcomings.

Today's current data becomes tomorrow's historical data, and future projects can use previous results to improve their estimating capability. Make sure your team members know how the data is being used. Give them access to the public metrics repository so they can view it—and use it—themselves. Remember to protect the privacy of individuals, though. One team member should never be able to view the personal data of another, although project- and organization-level data should be visible to all members of the project team.

Trap #9: Lack of Communication and Training

Symptoms: You may be falling into this trap if the participants in the metrics program don't understand what is expected of them, or if you hear a lot of opposition to the program. Fear of measurement is a classic sign that the objectives and intent of the program need to be better communicated. If people do not understand the measurements and have not been trained in how to perform them, they won't collect reliable data at the right times.

Solutions: Create a short training class to provide some basic background on software metrics, describe your program, and clarify each participant's role. Explain the individual data items being collected and how they will be used. Defuse the fear of measurement by stressing that individuals will not be evaluated on the basis of any software metrics data. Develop a handbook and web site with detailed definitions and procedures for each of the requested data items. Top-down communication from man-

agement should stress the need for data-driven decision-making, and the need to create a measurement-friendly culture.

Trap #10: Misinterpreting Metrics Data

Symptoms: A metric trend that jumps in an undesirable direction can stimulate a knee-jerk response to take some action to get the metric back on track. Conversely, metrics trends that warn of serious problems may be ignored by those who don't want to hear bad news. For example, if your defect densities increase despite quality improvement efforts, you might conclude the "improvements" are doing more harm than good, and be tempted to revert to old ways of working. In reality, improved testing might well find a larger fraction of the defects that are present—this is good!

Solutions: Monitor the trends that key metrics exhibit over time, and don't overreact to single data points. Make sure you understand the error bars around each of your measures, so you can tell whether a trend reversal is significant. If you can figure out why, say, your post-release bug correction effort has increased in two successive calendar quarters, you can decide whether corrective action is required. Allow time for the data you're collecting to settle into trends, and make sure you understand what the data is telling you before you change your course of action.

Requirements for an Effective Metrics Program

While individuals and project teams can easily apply a variety of metrics to track and improve they way they work, a successful organization-wide metrics program has several additional prerequisites. Most important is management commitment to creating a measurement-driven culture. The metrics planners must select a manageable, balanced set of relevant metrics and write clear definitions of the various data items being collected. Both the expectations and the results of the program must be clearly communicated to the participants, using training to explain the program and defuse the fear of measurement that is so common among software developers. Integrate measurement with your software process improvement program, using the latter to drive the desired behavioral changes, while relying on the metrics program to monitor results. Finally, you can evaluate individuals on their willingness to participate in the program, but never on the basis of the data they report.

Many of us struggle with how to implement a sensible metrics program that gives us the information we need to manage our projects and organizations more effectively. Staying alert to the ten risks described here can increase the chance of successfully implementing a software metrics initiative in your organization.

Bibliography

Austin, Robert D. *Measuring and Managing Performance in Organizations*. New York: Dorset House Publishing, 1996.

Daskalantonakis, Michael K. "A Practical View of Software Measurement and Implementation Experiences Within Motorola," *IEEE Transactions on Software Engineering*, vol. 18, no. 11 (November 1992), pp. 998-1010.

DeMarco, Tom. *Controlling Software Projects*. Englewood Cliffs, N.J.: Yourdon Press/Prentice-Hall, 1982.

Grady, Robert B., and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, N.J.: PTR Prentice-Hall, 1992.

Hetzel, Bill. *Making Software Measurement Work*. Wellesley, Mass.: QED Publishing Group, 1993.

Lorenz, Mark, and Jeff Kidd. *Object-Oriented Software Metrics*. Englewood Cliffs, NJ: PTR Prentice-Hall, 1994.

Rubin, Howard. “Measuring ‘Rigor’ and Putting Measurement into Action,” *American Programmer*, vol. 4, no. 9 (September 1991), pp. 9-23.

Wiegers, Karl E. *Creating a Software Engineering Culture*. New York: Dorset House Publishing, 1996.

New Software Concepts: Are Any of Them REALLY Breakthroughs?

Robert Glass, *Computing Trends*

Abstract

Every few years another software "breakthrough" is proclaimed. Each new breakthrough—object-orientation, cleanroom, formal methods, maturity models, and more—is said to provide an order of magnitude improvement in our ability to build software.

But invariably something goes wrong. Practitioners avoid or resist using these breakthroughs. Is it because practitioners are stubborn, or the breakthroughs are really BS? This contrarian talk explores the research findings that can help us answer that question.

Robert L. Glass *is the President of Computing Trends, publisher/editor of the* Software Practitioner*, and has been active in the field of software engineering for over 40 years, both in industry and academe. He is the author of over 20 books (many of them humorous) and 60 papers, and a columnist for several periodicals including Communications of the ACM. He is editor-in-chief of the* Journal of Systems and Software.

Coyote Valley Software Consulting
email: lawrence@acm.org

REQUIREMENTS HAPPENS…

This article originally appeared in the April 1997 Issue of *American Programmer.*

"I'm telling you it's got to load fast, and it has to have the Net siphoning tool!" exclaims Dale. "Maybe, but its real memory footprint can't be a bit larger than 14 ½ megs," replies Luke. "You two are always haggling over the wrong things," complains Jean, a frustrated marketing manager. "What we really need is the WebWarp™[1] feature, by no later than September."

Eight months later.

Dale: "We really *must* have the Net siphoner!"
Luke: "It'll push the footprint over 14 ½ megs! We can't do that!"

Two months later.

"I'm sorry Dale, but I'm canceling this project." explains Dana, VP of R&D, "We feel that we've missed the market window, now that Hype-A-Web™[2] is out, and we can't come up with anything that can compete in any reasonable timeframe."

You might think that this project was canceled due to management's inability to support its engineering staff adequately, or you might think it's analysis paralysis. I assert that the *real* reason was an inadequate requirements process.

What are we talking about?

It's hard to even talk about what requirements are when there's so much ambiguity in the term "requirements" and so many different ways that it's used. The *Random House Dictionary* says a requirement is "that which is required; a thing demanded or obligatory." The IEEE Standard 610.12-1990 says it's a "condition or capability needed by a user to solve a problem or achieve an objective." Then it lists two other definitions. The third one is very popular; "A documented representation of… (definition 1)."

While these definitions have their uses, let's consider a much broader one. A requirement is "anything that drives design choices." There are two main strengths in using such a broad definition. First, it encompasses a wide range of common reasons that projects fail, and second, it emphasizes that the requirements process—the process of collecting and applying the drivers of design choices—always happens, on every project, every single time. You may not always be aware that you're participating in a requirements process, though.

Using this definition of requirements, all kinds of things qualify as requirements that most other definitions wouldn't include. Your programmers' sense of aesthetics. Your senior management's prejudices about implementation platforms. Your design skills and experience. All these things drive design choices, as do how much money you're willing to spend on development, marketing, and sales. Of course, all the usual sets of requirements qualify as well. Your customers' expectations, stated and unstated. Their wants and needs. Lists of system functions, constraints, attributes, and so on. You have to take into account *all* of the requirements to achieve genuine success. Otherwise, they come back to haunt you. Every single time.

Requirements failure is a *BIG* problem.

My friend and colleague, III,[3] claims that "if you look at all the resources used on a software project, for the life of the system, about 70 percent of the total is spent on requirements."[4] Does that seem high?

[1] Just kidding. WebWarp isn't trademarked, so far as I know. What kind of product would it be if it was?
[2] See previous footnote.

 © Brian Lawrence, 1997. All Rights Reserved

Most projects I've encountered claim about 5 to 10 percent of the resources are *officially* allocated to requirements. What those estimates don't account for is all that other time spent arguing, persuading, haggling, and fighting over design choices. Have you ever participated in or witnessed the following dialog:

> *Tester: "It doesn't work like it should! This is a bug."*
> *Programmer: "That's the way it was designed. It's not a bug!"*

That conversation time and all of the time taken to discover, understand, and document the phenomenon in question are parts of the requirements process. And what about all those debates over which features to keep and which to throw away? How about the changing "standards" governing which bugs to fix and which to leave in your product as the ship date gets closer? All this time should be chalked up on the requirements clock. My experience in the software business confirms III's assertion; we spend well over half of our overall time in requirements-related activities. I've even had a client tell me outright that for their organization, the figure is closer to 90 percent — and the entire collection of project managers in the room agreed without hesitation.

We spend a lot of time — the majority of total project time — not implementing or testing, but trying to decide what to build and arguing over why and how. There's quite a collection of documented evidence[5] that requirements defects are by far the most expensive to fix once they propagate into designs and products. In my experience with project retrospectives, I've found that around 80 percent of serious defects are attributed to poor requirements.

If dealing with requirements takes the most time and produces the worst problems, then why don't we concentrate more on doing it better? There's more gold in this quality mine than in any other.

Why are better requirements so hard to get?

There are many barriers to better requirements. Here are some of them:

We aren't aware there's a problem. If we don't know we're engaging in a requirements process, how can we know how effective it is? Many organizations have real difficulties performing things like project retrospectives,[6] in which they'd find out just how much trouble their processes are giving them. I've heard a VP say, "We aren't holding a post-mortem because we don't want to dwell on the negative," just after canceling a project. That's precisely the time when a project retrospective *should* be held! Project retrospectives can be depressing if performed unprofessionally or with the wrong attitude. But they're probably the most powerful instrument for effective change in organizations that can support them. To solve a problem, you must first recognize that it exists. Learn to perform effective, "safe" project retrospectives.

We believe we know better. Humorist Will Rogers once said "It isn't what we don't know that gives us trouble; it's what we know that just ain't so." When we're all-knowing, we miss what others have to offer. The hardest question to answer is the one that is never asked. We know that most other people have a problem with the requirements process, but not us. We're the ones who know how to develop software. We "know what the user needs."

We don't know what else we can do. You can't do what you don't know how to do. Many organizations have simply never been exposed to more formalized approaches to requirements management. It's useful to view the requirements process as essentially a *negotiation* among the stakeholders over the value of whatever you're building.[7] It's not just that you may not know what

---

[3] III is pronounced "Three", and is his full, legal name.

[4] Actually, III uses the term "essence modeling" among other things to refer to what I'm calling requirements.

[5] See Barry Boehm's *Software Engineering Economics* [1], p. 40, for the classic reference.

[6] I'm using "project retrospective" in place of "post-mortem, post-partum" and other names for post-project analyses.

[7] Tom DeMarco did an excellent tutorial on requirements as a negotiation process at ICRE'96[2].

 © Brian Lawrence, 1997. All Rights Reserved

else to do. You probably don't have the skills, either. Most programmers are selected for their ability to manipulate code, not for their communication and negotiation skills. These skills can be learned, however.

"It's not our job." Marketing says it's R&D's responsibility. R&D says it's Marketing's. Everyone agrees that it's the customers' responsibility. Nobody's talking to the customers. The simple fact is that in the absence of any other input, it's the developers who *must* take responsibility for managing requirements. They're the ones who need to know what to build. In the most effective requirements processes, *every* important stakeholder has a role.

We know about it, but we don't have permission to fix it. Senior management says "Just do it right this time!" then proceeds to evaluate their staff using the same old criteria, which discourages taking chances on new processes. Senior managers say they want it, but they don't provide funding for training and consulting or time in the schedule to learn new techniques. Lip service without the resources and policy changes to support the changes can be *very* encouraging — encouraging your best staff to find some other place to work, that is. Organizations that fall into this trap are suffering from an inadequate *chartering*[8] process.

Our organization is so dysfunctional, we can't fix anything, even if we wanted to. You're forever behind the eight ball. You don't have enough time to get your regular work done, much less try out these newfangled ideas. There's no way in the world you're going to take on any new requirements process. If you're in this situation, things are tough. Maybe you should find another job. If you've decided to hang in there, then wait until the right moment to consider alternative processes. Before the start of a new project is a good time to introduce a revised requirements process.

We know about it, have permission and resources, but golly, this is tough! Even in the best of circumstances, requirements management is hard work. You need to make tough decisions. Lots of people need to be involved, but some of them won't see it as their responsibility to participate. Customers have been telling you misleading things. It's hard to maintain the level of energy and the focus to sustain your momentum. It's important to recognize that managing requirements is hard work; that's one big reason why we spend so much time doing it badly.

Notice that I didn't say "Requirements are hard to nail down because they're constantly changing." I don't believe that. The real problems customers face don't change all that fast. What changes is our perceptions of the problems. If we can work our way past all the obstacles that blur and block our views of our customers' problems, then requirements can be clear and stable.

What are some possible approaches?

Here I present some potential approaches to managing requirements, roughly in order of popularity. I want to emphasize that all of these approaches work just fine *on some class of problems*. Difficulties arise when a particular approach is used on the wrong class of problem. And the difficulties that come up are often unpleasant ones: missed market windows, hopelessly flawed products shipped, and canceled projects, whose only output are burned-out, dispirited people.

Get an idea, then build towards the vision it creates. "If we can get that Net Siphoner out by June, we'll sweep the market!" We like following an idea because we know how to do it, and it feels natural. A lot of software is designed this way. And it works, but it's risky.

Feature wars. "They have Web Conferencing! If we don't have it, the magazine reviewers will rate our product second best." Comparing features and competing by offering more, newer, "cooler" features is another immensely popular approach to managing requirements. It has sold a lot of software.

Analysis. There are a whole host of modeling techniques that make up the field of analysis. Data flow diagramming, entity relationship modeling, data dictionaries, state models, event models, object

[8] A charter is a contract between the purse string holders and the doers; it contains a system boundary, objectives, commitment of resources, and a designation of who approves the work.

 © Brian Lawrence, 1997. All Rights Reserved

oriented models, decision trees, use cases, and so on. All are good techniques for modeling some aspects of customer needs. Analysis consists of many highly effective techniques for better understanding both requirements and design.

JAD. Both Joint Application Development and Joint Application Design[9] are outgrowths of participatory design approaches used at IBM starting in the 1970s. The essential idea is to bring people with potentially important contributions together in a session where they can contribute their ideas and come to agreement on requirements and design. JAD sessions typically have designers, management, and customers all working together using a facilitator, visual aids, and a scribe to generate specifications in a relatively short amount of time.

Quality Function Deployment (QFD). W. Edwards Deming and Harold Dodge went to Japan in the '50s, and one of the things that came back is QFD. QFD was originally devised by Yoji Akao in Japan's Kobe Shipyards, where Akao was trying to find a way to get everyone to share responsibility for quality. In QFD, teams construct tables to map customer benefits to design criteria, and design criteria to features. QFD is a family of methods that bring business goals and customer perceptions into your development process in a visible, controlled manner.

Soft Systems Approach. The soft systems approach to requirements management, described in *Exploring Requirements* [3], uses a series of heuristics to develop and gain consensus on both customer problems and designer considerations. Like JAD and QFD, it uses teams to develop the requirements information in facilitated meetings. It has the advantage of possessing models for every kind of requirement covered by my definition. It's designed to stimulate innovation while discovering exactly what your customer needs.

Formal Methods. Formal methods use specification languages (such as "Z") or diagramming models which are typically mathematically well-defined. Formal methods have been used on some large systems development, and they are popular in academic settings. They are especially good at identifying internal inconsistencies in requirements.

All of these approaches have at least one thing in common; they are ways of modeling customer needs. Some do it by modeling potential solutions ("Get an idea," "Feature wars"), while virtually all the others model the problem.

Why model requirements? Or anything for that matter? Modeling serves many purposes, three of which are:

1. It allows us to gain a better understanding simply by the act of constructing the model.
2. It provides a basis for testing what we want to build to see if it matches reality.
3. It enables us to share our understanding of the design problem with other stakeholders.

Modeling has another critically important attribute: it makes the subject visible. So many of the reasons we choose to build what we do are hidden away, locked up in the minds of important participants. Users' perceptions. Programmers' preferences. The number one enemy of software projects is hidden, false assumptions. If there's one thing a requirements management process should do, it's to bring those assumptions into the light of day, so they can be examined, tested, and verified.

Hard problems demand significant commitment.

Have you ever gone downhill skiing? If you're a good skier, remember what it was like when you first started out. Imagine this scenario. You don all the fancy equipment, then you hit the slopes. The first time you try to stop, you *literally* hit the slopes: you fall down. After some practice on the baby slope, you're getting pretty good at turning and stopping. (You have that hockey stop down ice cold, and you've figured out how to shower your friends with flying snow so they're ice cold, too!) Then some idiot friend talks you into trying the expert run, starting way up at the top. You fall off the lift at the beginning of the run. This is your first indicator that this wasn't such a good idea.

[9] JAD® representing "Joint Application Design" is a registered trademark of IBM Corporation.

 © Brian Lawrence, 1997. All Rights Reserved

You've managed to get down in one piece, but you're miserable and exhausted. That part of the mogul field directly under the lift was particularly humiliating. Falling down all those times and having to crawl back up the snow to recover your skis, while all those people were watching and laughing; you wished you could have died. At least you're OK. Your friend was hauled away by the ski patrol and will be sporting plaster for a few months.

I've seen a few software projects that went pretty much like that little adventure. You set out thinking it'll be fun and successful. Then reality sets in. You're taking on the double diamond run with baby slope methods. What you need are really good lessons.

Exceptional ski instruction programs[10] have some defining characteristics. They form groups with mostly the same skill level and keep the group together long enough to learn successfully. They talk about theory and also show you technique. Then they make you do it. Over and over. The instructors take you over terrain you wouldn't choose to go on by yourself. They film you, then show you yourself, critiquing your technique. All in a "safe" environment. When you get into trouble, they're there to help you so you don't get too discouraged.

Learning better requirements techniques is a lot like learning to be an expert skier. Most ski schools can get skiers to advance from beginner to intermediate relatively easily. But advancing beyond intermediate level gets much harder. Moreover, with a requirements process, you aren't just trying to get a single individual to learn; you have to teach a whole team, some of whom don't especially want to be there.

Good skiing lessons aren't cheap. Better requirements aren't either. You need more than training. It really helps to have someone along who actually knows what he or she is doing. That seems like a simple observation, but an awful lot of people overlook it. I confess that I overlooked it until I heard Deming point it out during a four-day seminar. In the same breath, he also said "Get the very best help you can possibly find." Sound advice. Please understand that I'm not necessarily advocating bringing in consultants. Good help can be found in a variety of places. For example, check out successful projects for people who might be able to join your effort. When you set out to improve your requirements management process, define a proper charter for the effort, get good training, *and* get some hand holding during the whole process. It makes all the difference.

The purpose of any requirements management process can be summarized by two questions:

1. Are we doing the right thing?
2. What makes us think so?

The degree to which your requirements process answers these questions is the degree of possible success your project and product might enjoy. How much you choose to invest to answer these questions should depend on the potential value in producing your product, together with the amount of risk you're willing to take. The potential return on investment for improved requirements management is very, very high.

So how do you choose which technique to use? It's hard to say. So much depends on how much risk you're willing to take and what might work in your culture. A good craftsman knows how to use the tools of the trade and when to use which tool. Both chainsaws and table saws are powerful and effective, but they're made to solve very different problems. Less formal requirements methods won't help you to understand your customer's needs as well as more formal ones will. Don't take risks unknowingly.

Requirements exist whether you identify them explicitly or not, and they profoundly affect cost, schedule, and market success. You have a better opportunity to control cost, schedule, and market success if you identify requirements explicitly. Consider this: you're likely already spending more on your existing requirements process than on any other activity. You may think it'll be expensive and risky to attempt to improve your requirements process. It's even more expensive and more risky not to. And if you think your "easy" problems don't require better requirements techniques, just remember: those expert skiers handle the baby slope pretty darn well.

[10] Such as the *Breakthrough Program* at Vail, Colorado.

 © Brian Lawrence, 1997. All Rights Reserved

Acknowledgments

I'd like to express my gratitude to III, Bob Johnson, and James Bach for their very helpful insights in discussing and reviewing this article. And thanks also to my other reviewers: Cem Kaner, Bill Pardee, Sharon Marsh Roberts, Melora Svoboda, Payson Hall, Tom DeMarco, and Don Gause.

References

1. Boehm, Barry, *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice Hall, 1981.
2. DeMarco, Tom, "Tutorial Notes: Managing the Requirements Process", In *Proceedings of the 1996 IEEE International Conference on Requirements Engineering*, Los Alamitos, CA: IEEE Computer Society Press, 1996.
3. Gause, Don and Jerry Weinberg, *Exploring Requirements*, New York: Dorset House, 1989.

Additional Reading

On Design

Alexander, Christopher, *Notes on the Synthesis of Form*, Cambridge, MA: Harvard University Press, 1979.

On the Journey of Creativity and Design (Life Cycle)

Koberg, Don and Jim Bagnall, *The Universal Traveler*, Menlo Park, CA: Crisp Publications, 1991.

On JAD

Wood, Jane and Denise Silver, *Joint Application Development*, New York: Wiley, 1995.

On Analysis

DeMarco, Tom, *Structured Analysis and System Specification*, Englewood Cliffs, NJ: Prentice Hall, Yourdon Press, 1979.

McMenamin, Stephen and John Palmer, *Essential Systems Analysis*, Englewood Cliffs, NJ: Prentice Hall, Yourdon Press, 1984.

On Quality Function Deployment

Pardee, Bill, *To Satisfy and Delight Your Customer*, New York: Dorset House, 1996

On Negotiation

Fisher, Roger and William Ury, *Getting to Yes*, New York: Penguin Books, 1991.

On Facilitation

Kaner, Sam, et al., *Facilitator's Guide to Participatory Decision Making*, Philadelphia: New Society Publishers, 1996.

 © Brian Lawrence, 1997. All Rights Reserved

On Really Understanding and Changing Organizations

Weinberg, Jerry, *Quality Software Management*, 3 volumes, New York: Dorset House , 1991-94.

Oshry, Barry, *Seeing Systems*, San Francisco: Berrett-Koehler, 1995.

Brian Lawrence is an author and consultant who teaches and facilitates requirements analysis, peer reviews, project planning, life cycles, and design specification techniques. Mr. Lawrence is serving as program chair for paper selection for the 1997 Software Engineering Process Group Conference and as the industrial program chair for the IEEE Computer Society's 1998 International Conference on Requirements Engineering. Mr. Lawrence is a participant in Jerry Weinberg's 1996 Software Engineering Management Group and is a member of the ACM and the IEEE Computer Society.

Mr. Lawrence can be reached at Coyote Valley Software Consulting, 335 Keeler Court, San Jose, CA 95139 (408/578-9661; e-mail: lawrence@acm.org; WWW: http://www.stlabs.com/lawrence/cvsc.htm)

 © Brian Lawrence, 1997. All Rights Reserved

WHAT BUGS THE SOFTWARE INDUSTRY? RESULTS FROM AN INDUSTRY SURVEY

Wolfgang B. Strigel
Software Productivity Centre
#460-1122 Mainland Street
Vancouver BC, V6B 5L1 Canada
Phone: 604-662-8181 ext. 101
Fax: 604-689-0141
email: strigel@spc.ca

http://www.spc.ca

1. Abstract

The Software Productivity Centre (SPC) conducted a survey of 34 software companies in BC with the goal to find answers to the following questions:

- What are the dominant impediments to successful software development?
- What are the critical skills gaps experienced by professionals in the industry?
- Which recommendations to industry and educational institutions can be derived?

Information technology is the largest industry segment worldwide. Software development is one of the most critical contributors to this industry. Our observations with more than 140 member companies worldwide indicate significant problems in this industry to cope with the increasing demand for software solutions and to keep pace with the rapid evolution of software technology. Our focus for this study was therefore to identify common problems in the industry and ways to address these issues effectively.

This paper summarizes the results from the survey. It shows how the survey results support our observations from over 40 process assessments. Interestingly, the most significant issue in the software industry is of managerial and not technical nature. The paper explores reasons for this situation and suggests solution approaches. One of the solutions is a proposed new concept for continuing education in software engineering and the management of software development.

2. Keywords

Software engineering, software industry, management, process assessment, process improvement, training, continuing education.

3. Executive Summary

The SPC survey was sent to 67 software companies of which 34 replied. It was entitled "Technology and Skills Gap Analysis". As such, its primary focus was to identify critical skills which are necessary for the success of a software company and which are not sufficiently available in the industry. The results also showed some well-known issues such as scheduling problems, controlling the requirements, shortage of qualified staff, etc. This is not a big news item for those who are working in this industry. However, we found it interesting that most respondents were quite satisfied with the technical skills of their staff and the technical expertise conveyed by educational institutions. The most significant gap was identified in management skills.

Many papers have been published about the so-called "software crisis". This crisis fueled the emergence of software engineering, which is gradually becoming a recognized discipline and has made inroads into university curricula. However, it is the thesis of this paper that software engineering principles alone are not sufficient to deal with current and future challenges. Well-known problems with schedule and budget overruns in conventional projects are now compounded by

- rapidly increasing volume in demand for more software
- dramatic shortening of the development life cycle
- significant acceleration of technological evolution
- increasing competition and the need for effective product positioning.

When combining the results from the study with SPC's experience from process assessments, we see the need to expand the focus from process improvement at the corporate level to an increase in the competency of the individual manager. The importance of the individual competence of software developers has been addressed in many publications ("Peopleware"[1], "Decline and Fall of the American Programmer"[2], productivity factors in the COCOMO[3] model) and more recently by the Personal Software Process[4] and SEI's People Maturity Model[5]. The main focus of these publications was on the capability of the programmer. This paper suggests that based on the results of our survey, another significant problem lies with those who manage the development process. Thoughts on how these management problems can be addressed effectively are presented in this paper.

4. The Survey

4.1 Study Approach

Our approach was to survey a representative sample of the software industry in British Columbia, Canada. A steering committee representing a cross section of software professionals was convened to guide the survey. Several focus group sessions were conducted to design the questionnaire. The companies selected for the study were classified into 4 sectors: software products, software services, system integrators and in-house software development (IS departments). Over 50% of the surveyed companies responded for a total of 34 completed questionnaires. These companies represent 2,086 software professionals.

The results from about 500 questions were analyzed with the help of a professional statistician. After an initial analysis of the data, we conducted 10 on-site interview sessions with a subset of the respondents and representatives from universities and colleges to explore possible recommendations. The results were compiled in a detailed report, which was published in April 1997[6].

4.2 Survey Results

This section presents the survey results at a high level. More details and a full report can be obtained from the SPC[6]. To interpret the survey results it is important to consider that the questionnaires were typically answered by people in technical management positions. This includes positions such as VP of Development or equivalent responsibilities. Figure 4.1 shows the distribution of respondents by company type.

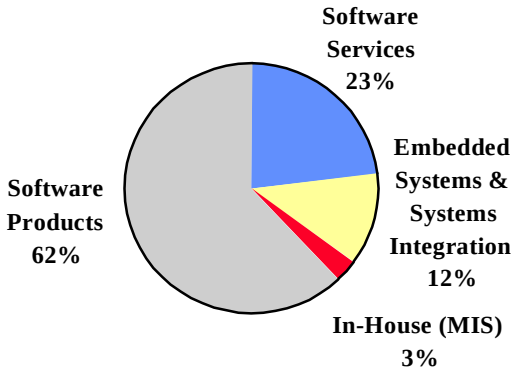


Figure 4.1 Survey Profile

The first major focus of the questionnaire was to identify overall industry concerns. The companies were asked to select the top four general (business related) issues. They are:

- Ability to recruit qualified people
- Achieving customer satisfaction (a quality issue)
- Managing software projects to meet schedule and budget
- Managing rapid company growth

Figure 4.2 shows details for all industry issues.

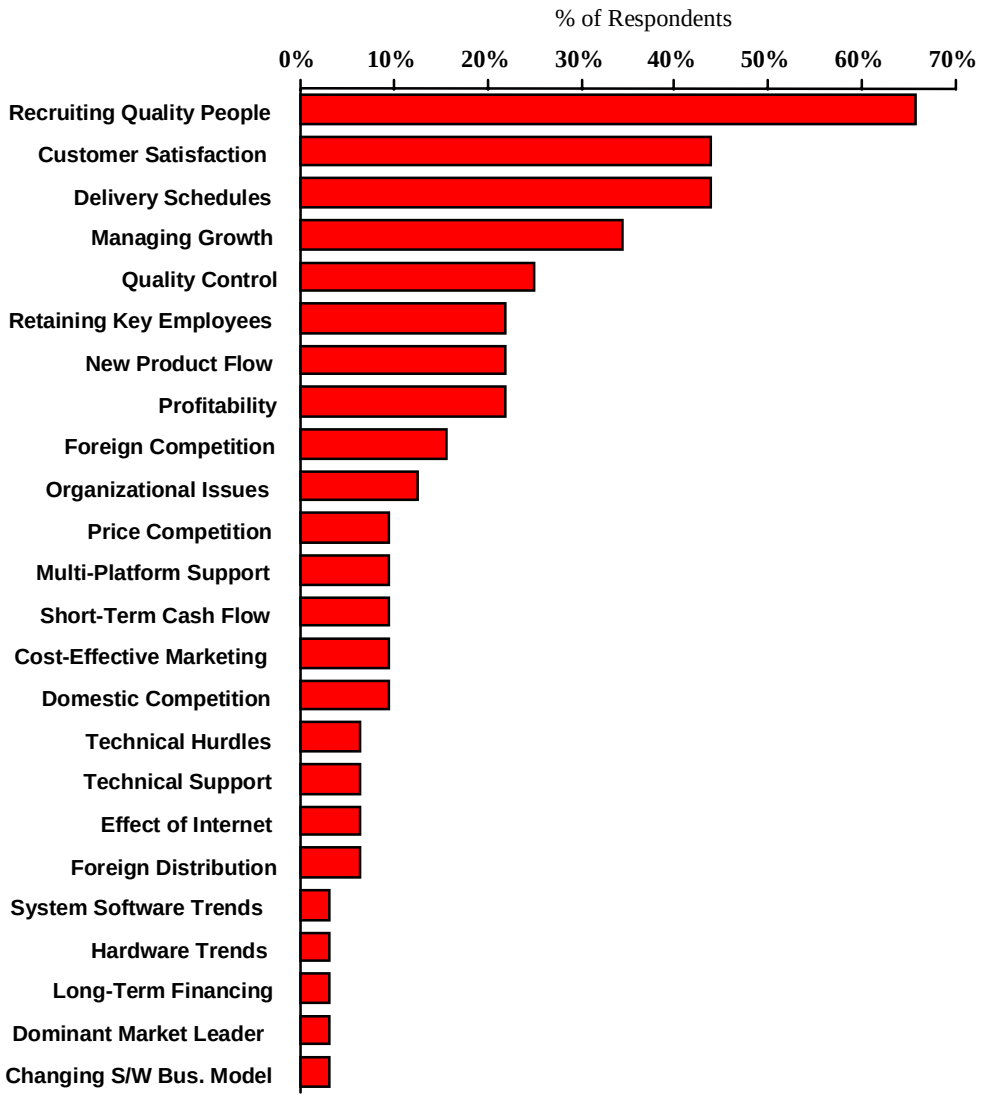


Figure 4.2 Industry Issues

To identify the skills gap, we then asked the respondents to rank their perception of current industry capabilities. Figure 4.3 shows the respondents' assessment of their core engineering capabilities.

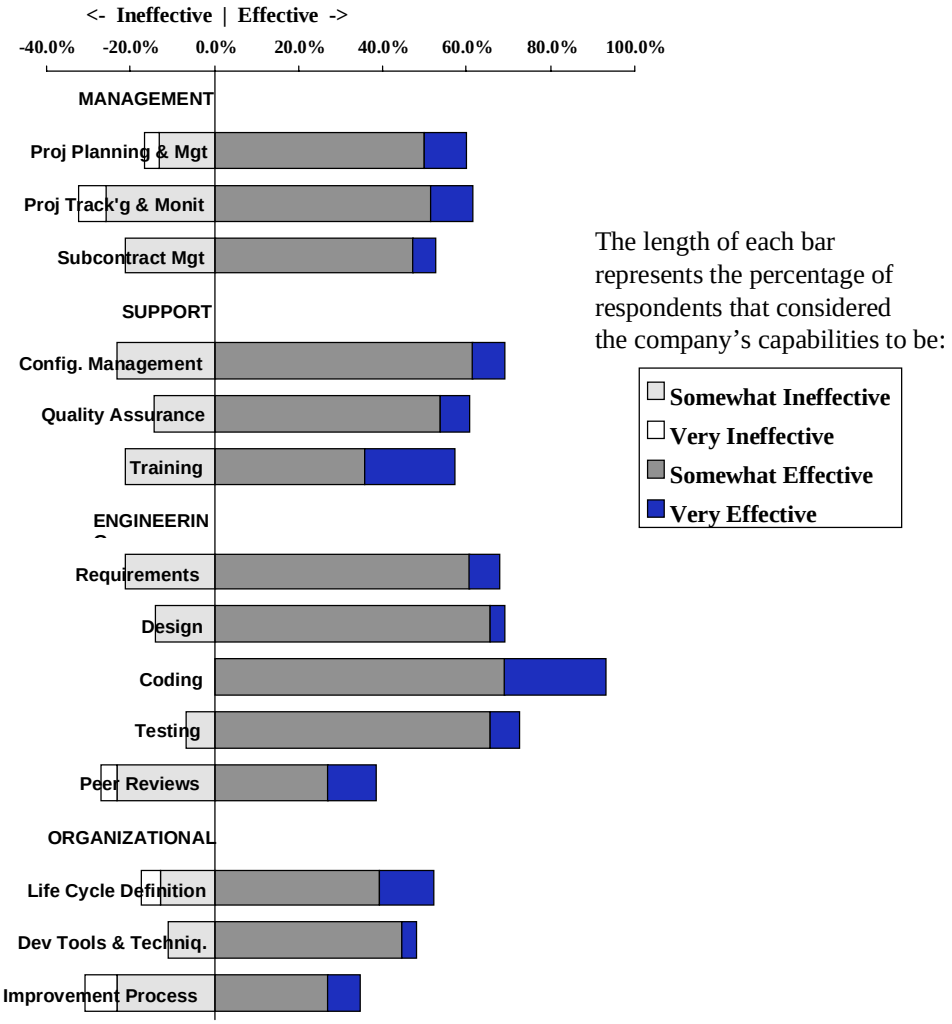


Figure 4.3 Industry Capabilities

Most responses ranged from neutral to fairly positive. We believe that these responses were on the optimistic side. This may be a result of a somewhat biased view, given that the respondents were mostly responsible for the categories in question. It tends to contradict some previously identified industry issues and some of the development issues shown below. Our assessment experience has shown that the reality is somewhat different. For example it is questionable if capabilities in project management and tracking are effective if, on the other hand, the number one development problem is "schedule overruns".

The next area of interest was related to key software development issues. The three most critical problems were:

- Schedule overruns

- Availability of skilled staff
- Poorly defined requirements

Figure 4.4 shows details for this category from the survey results.

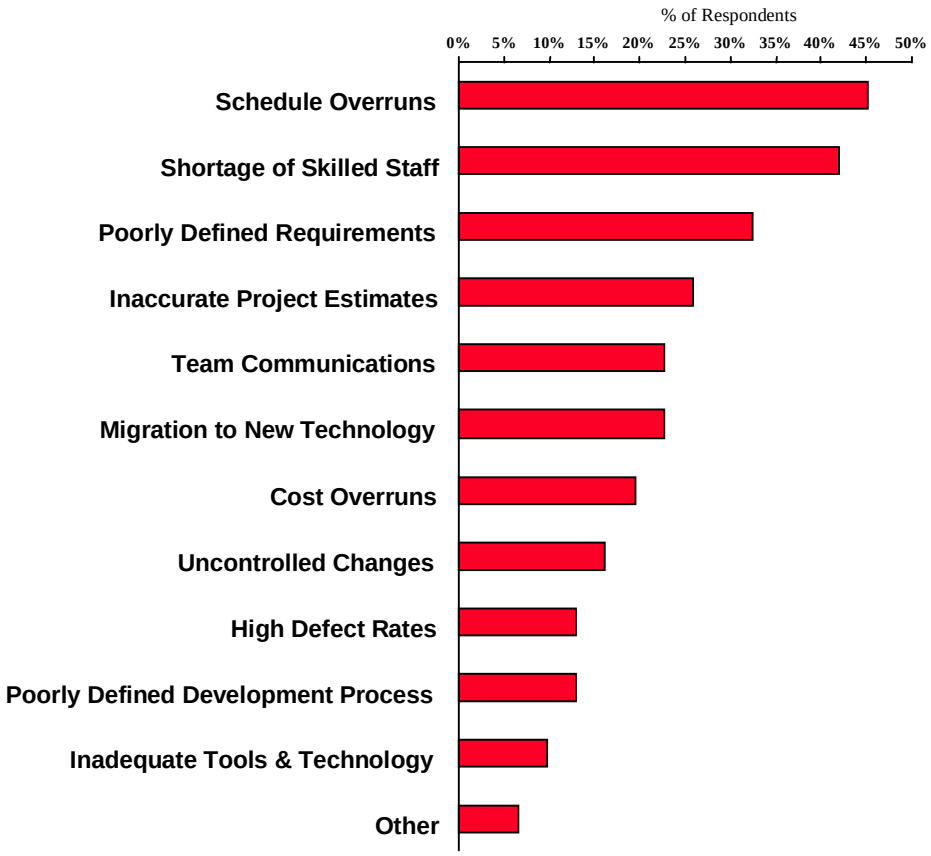


Figure 4.4 Software Development Problems

We were also interested to learn more about the impact of the acceleration of changes in technology. The half-life time of current technical knowledge is anywhere between 6 months and two years, depending in which application sector a company is active. In the Internet market, for example, languages, tools and generally available products tend to evolve extremely fast, whereas other applications have longer life expectancy. Even the traditionally slow moving area of information systems applications is currently undergoing significant change with the introduction of Intranets, client server technology, and year 2000 issues. Figure 4.5 shows the details.

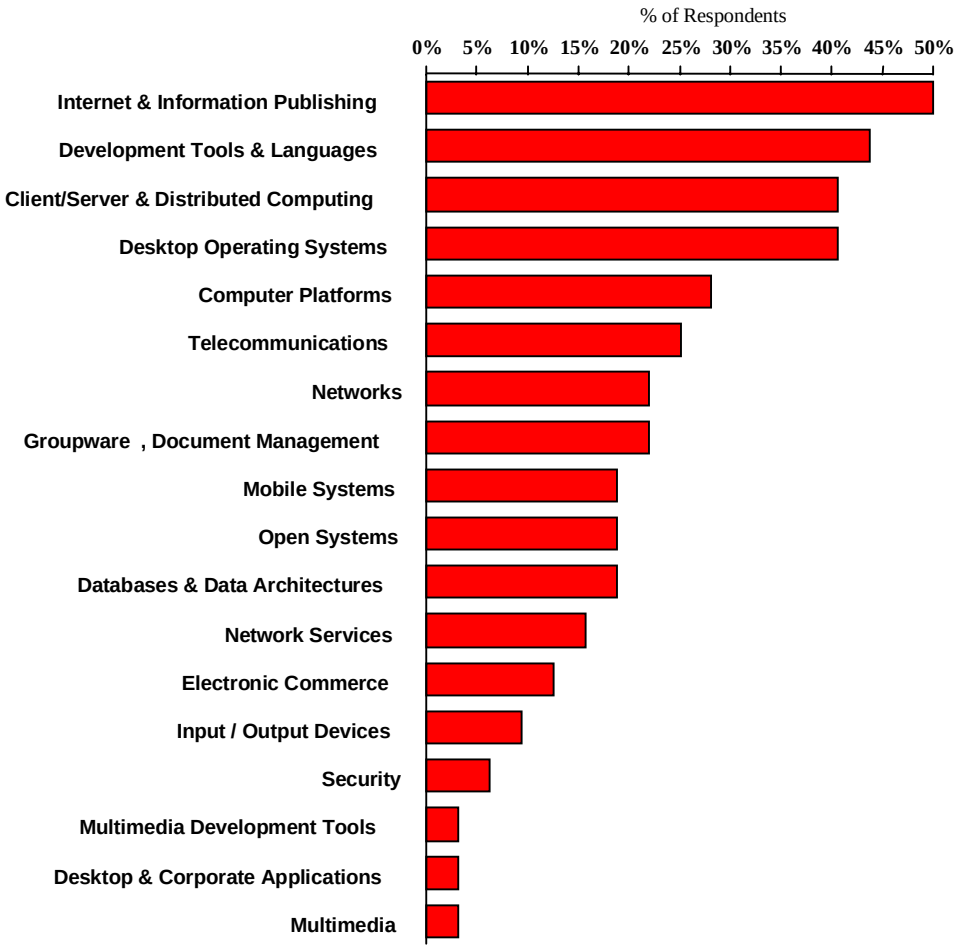


Figure 4.5 Technology Change Concerns

Another view of technology issues was given by the responses to questions related to current and future technical and non-technical needs, which are shown in Figure 4.6.

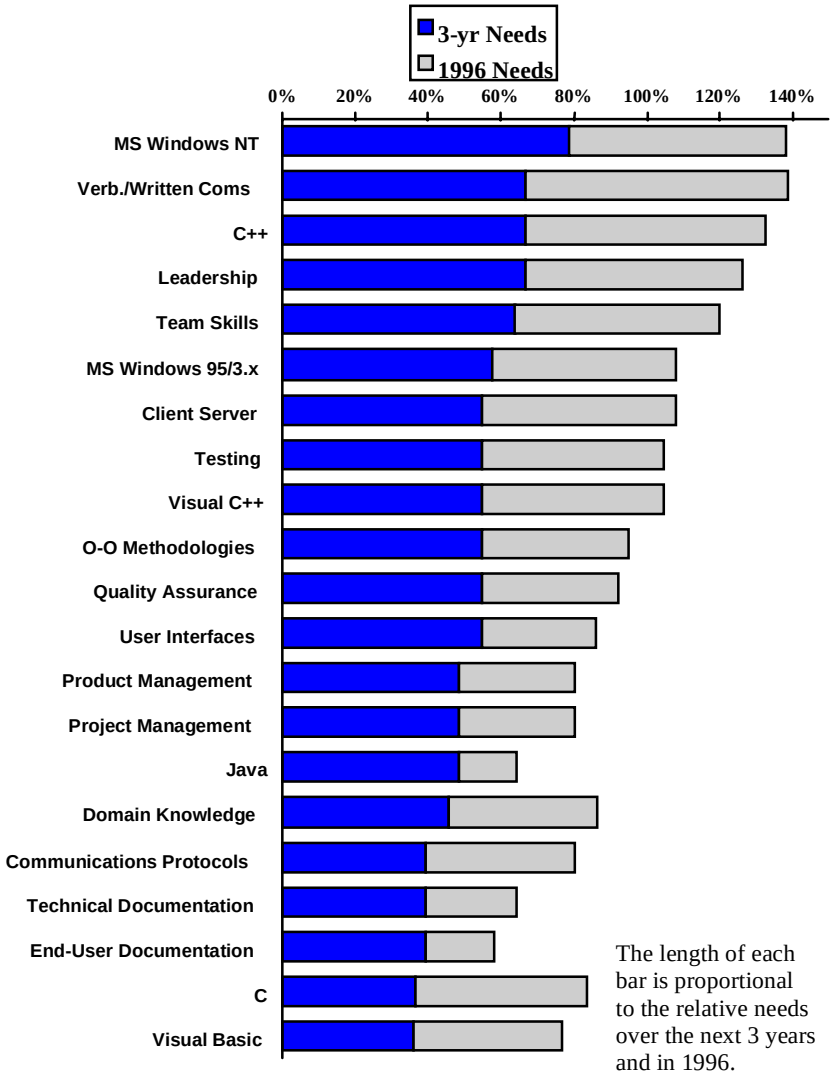


Figure 4.6 Technical and Non-Technical Needs

For the purpose of comparing technical needs with non-technical ones (especially in the area of management and soft-skills), the diagram shows these non-technical needs as well. It should be noted that the significant gap in domain knowledge is not further analyzed. The issue of how to foster application domain knowledge is outside the scope of this paper.

The final area of interest was training. Our focus was to correlate training practices in the industry with the perceived skills gaps. The following figures show a summary of the survey results in this area.

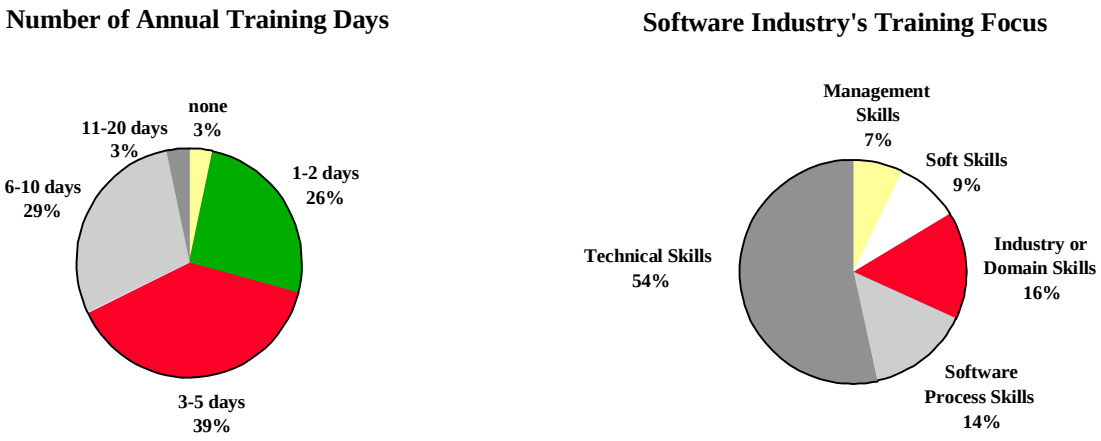


Figure 4.6 Industry Training

Further training related questions focussed on the effectiveness of various training delivery methods. Company internal training ranked on-the -job training as the most effective method, followed by in-house courses and vendor supplied courses. Computer based training and video based training received the lowest effectiveness ranking.

5. Analysis of Results

It can be argued that the sample of surveyed companies is limited to BC and the observations might not apply to other areas such as Silicon Valley for example. However, our results seem to echo similar observations reported in other papers and books on the topic. In addition, our analysis of the survey results is combined with observations from CMM-type process assessments that were conducted by the SPC over the last 5 years[7] and which covered companies across North America. It also reflects our experience in consulting with many of the 140 member companies of the SPC over this period.

The results are analyzed along the following 3 dimensions:

- Technology
- Process
- Soft skills.

The latter two categories will lead to recommendations in the general area of management which, in our opinion, is a combination of process knowledge and soft skills.

5.1 Technology Related Results

It is interesting that under ‘general industry concerns’, the technological hurdles rank in 16th place with only 7% of all responses. Similarly, in the category of ‘software development problems’, the item ‘inadequate tools and technology’ ranked 11th with 10%

of all responses. Conversely, in the training category, the highest degree of satisfaction was with programming and technology skills as they are conveyed by educational institutions.

Considering the high degree of technical complexity of software development, these are interesting results. Of course some concerns are expressed in the items under 'technological change'. After all, we are experiencing a dramatic acceleration of the introduction of new technologies, primarily related to the Internet. The general flavour of the survey results, however, does not indicate a major problem with that. The technical skills, which rated highest, were topics related to immediate needs such as knowledge of MS Windows or C++. These are the burning issues for most technical managers since the need in this area significantly outweighs the supply. Most of the position openings that we are aware of, are for people with these skills. In a way, this is not bad news, since such skills can be acquired relatively quickly, provided that the individuals have a good understanding of the underlying technical concepts.

Overall, it seems that the survey participants felt fairly comfortable with the availability of technical skills. We believe that this result is somewhat influenced by the background of technical managers who responded to the survey. It may reflect some degree of unjustified optimism, which may not be shared, for example, by company presidents. On the positive side it reflects the very strong technical capability of the companies surveyed.

5.2 Process Related Results

The term process is still not in wide spread use and in some instances it is not clearly understood. Apart from a few questions, it was not explicitly addressed in the survey. In this paper, we refer to process as the application of methods, tools and procedures used to control software development activities. A special section is dedicated to this issue because many problems identified in the survey are indicative of process issues. For instance, in the category of 'industry issues' (Fig.4.2) we obtained:

- Problems with delivery schedules, ranked in 3rd place at 43%
- Managing growth ranks 4th at 35%
- Quality control ranks 5th at 25%.

Under 'industry capabilities', the assessment of process improvement capability obtained one of the worst effectiveness ratings. This happens to have a strong bearing on the ability to manage growth. The good news is that a majority of the surveyed companies are successful and are experiencing significant growth. Along with growth come growing pains which are more pronounced in organizations that lack a defined development process. Similarly, under category of 'software development problems' (Fig. 4.4), some of the process related issues were rated as highly problematic:

- Schedule overruns ranked 1st place at 45%
- Poorly defined requirements ranked 3rd place at 33%
- Inaccurate estimates ranked 4th place at 25%.

All of these issues can be directly mapped to the SEI CMM level 2. Not surprisingly, they also echo our observations from process assessments[7]. In our estimate, none of the responding companies would be rated higher than SEI level 2. In an industry that is experiencing an explosive growth in demand, companies with weak internal processes can survive as long as the growth exceeds supply significantly. As soon as the demand enters a period of diminished growth, these companies are more likely to fail than their competitors with stronger process capabilities.

5.3 Results Related to Soft Skills

It is generally recognized that soft skills are not among the strong points in the skill set of the typical technical expert. This is obviously a generalization but we know from experience, and from company internal psychological profiles, that knowledge workers are often weak in interpersonal communications, people skills, and hence in some of the fundamental prerequisites for a successful manager. These observations are confirmed in the results from our survey:

- Needs for verbal and written communications rank in 2nd place (Fig. 4.6)
- Needs for leadership skills rank in 4th place (Fig. 4.6)
- The 5th most significant development problem is with team communications (Fig. 4.4).

Most of these skill gaps are ranked higher than the need for Microsoft Windows 95™ skills. That was probably the most amazing result of our survey. But again, judging from our consulting experience, it is a correct reflection of a recurring weakness in software development companies.

5.4 Summary of Survey Results

The most significant result of our survey is that process and soft skills are identified as much bigger problems than technical skills. Unfortunately, the soft skills are arguably more difficult to train than the technical skills. The problem is compounded by the almost complete absence of soft skills training in the typical university or technical college curriculum. Process skills are slowly introduced at universities in the context of software engineering courses, but the coverage of software engineering at most universities is still far from adequate.

The economic impact of these results should not be underestimated. The most significant impediment to industry growth is related to the ability to staff projects with experienced people. According to the survey, 7.1% of the total positions remain unfilled. This may not appear to be a large percentage, but most of these unfilled positions are in the area of management and senior technical expertise. If these positions were filled, an even larger number of developers would be required to staff the projects. For Canada, government statistics for 1994 show a total employment of about 208,000 software workers[8]. Seven percent of this number represent 14,500 open positions. Using a general rule of thumb of

$100,000 gross revenue per employee in the software industry, this represents $1,450,000,000 in lost revenue in the Canadian software industry. It could be argued that a significant portion of this number is attributable to the lack of software development managers.

6. Solving the Management Crisis

Management positions exercise significant influence over the outcome of projects. There are several critical management roles in the typical software development organization, including the CEO, the VP Marketing, the product manager and other management roles in finance and support. In the following, we will concentrate on the positions responsible for software development, primarily the development manager or equivalent. The considerations include various seniority levels from team leader to VP of Development.

6.1 Example of a Fast-track Career

The phenomenon of poor management is not surprising if we look at the typical career path of the average development manager (let us call her Lucy). Lucy went through academic training in engineering or computer science. Throughout this training she never took a course in communications, finance, marketing, product management, team management, or similar. She knows, however, a lot about formal languages, computer architecture and so on. Having taken several courses in data base design, she is hired by a company that develops data base middleware products. After a one-year stint in testing (that is where un-experienced people are initiated to the real world), she moves into development and quickly gets recognition for her ability to generate lots of code in record speed. After 2 years of increasing design responsibilities, she is promoted to lead a team, which develops a new application. As the project runs into trouble, her team is increased and within six months she finds herself heading a team of eight programmers. Of course there was never any time to train Lucy in management skills. After all she has proven a high degree of competence as designer and is intelligent enough to learn the use of some project management tools on the fly (and nobody notices that she uses the project management tools merely as a drawing tool for Gantt charts).

Lucy's company experiences major growth. As a matter of fact the company grows 50% annually. This means that every two years the need for managers doubles. Since Lucy continues to impress management by her optimistic attitude (she never disputes schedules imposed by marketing) she is quickly promoted to become a section head. In the meanwhile several release dates had to be shifted and the manager of development is taking the heat for the delays. After the loss of a major OEM deal the development manager is fired. Lucy, with her helpful attitude, is the obvious candidate to replace him. After 5 years of industrial experience, she is now responsible for the development department in a 50 people company with 20 developers. During her career she only had time to take several courses in advanced data base technology and object oriented design.

This fictitious case is very typical according to our assessment experience. Lucy was fast-tracked and she may have the right attitude for senior management positions (smart, assertive, extrovert). But she was never given the training and understanding in management tools and techniques. The speed of the career path varies depending on the industry. Product development companies offer the fastest career progression potential, followed by system integrators and finally by internal IS organizations. This mirrors the typical growth rate for the respective industry segment. The symptoms, however, are similar in each case. One of the factors, which contribute to this problem, is that the growth of companies in our industry often outpaces the ability of individuals to grow into management positions[9]. To cope with this growth there are only two solutions: fast-track internal staff or hire managers from the outside.

The other problem is that there never seems to be time to train people in management skills. As the survey shows, training in technical topics receives most of the training dollars. This is understandable since it addresses short-term needs (e.g. Microsoft Windows, C++, etc.) which have to be dealt with. Their benefits are immediate.

6.2 The MBA Phenomenon

In the 1980's an MBA (Master of Business Administration) became a very sought after and a very fashionable degree. Other than prestige, it promised an immediate and significant increase in compensation potential. It also offered other lasting benefits.

MBA programs are mostly non-industry specific. They are designed to teach fundamental principles of management, economics, finance and business strategy. Most MBA curricula are designed for traditional manufacturing applications and mass market products.

MBA programs can be taken directly after completing an undergraduate degree. In our opinion this has limited use since the student has not yet acquired the industrial experience which allows him or her to relate the material to real life industrial situations. The more useful MBA model is that of an "Executive MBA" which caters to mature students with several years of industrial experience. These programs can be taken during evening or weekend classes and allow the student to continue his/her duties in the work place.

It is interesting that the MBA concept experienced most of its success in traditional industries (predominantly in manufacturing and finance). In comparison to the software industry these sectors are characterized by a relatively slow change of business environment. Why are there no MBA programs for the software industry?

6.3 The Case for an MBSA

The concept of continuing education is now common place for most people in the work force. In software development, where the half life time of knowledge in shrinking every

year, the typical continuing education focus is on training new technical concepts. Given the shortage of management talent in the software industry, it would be extremely useful to have a continuing education curriculum for those aspiring to climb the management ladder in our industry.

This paper therefore proposes the creation of a degree in software management, or MBSA for "Masters in Business of Software Administration". This terminology was chosen deliberately. Software development is not only a scientific undertaking. It is the use of sophisticated technical concepts to build complex products for broad markets. It frequently has an enormous impact of the quality of life and the safety of just about anything we operate. It stands to reason to have highly qualified managers responsible to generate these products. Information technology is the biggest single market segment world wide, bigger than agriculture, natural resources, the car industry etc. The global software market alone will exceed $512 billion by the year 2000[10].

Several attempts have been made to create similar programs. Some leaders in our industry, including Barry Boehm and Tony Wasserman (from private correspondence) have tried it without success. One of the problems is that such a curriculum would require the collaboration of several university faculties (namely Business Administration and Science). Such cross-faculty collaboration seems to be fraught with problems. However, as our survey demonstrates, there is a need for such a program and wherever there is a demand, supply will eventually be forthcoming (according to Economics 101).

The delivery of the MBSA could be strongly biased towards distance education. After all, the target audience is conversant with electronic media and an Internet based delivery for a good part of the program should be feasible. It is beyond the scope of this paper to explore the details of an MBSA curriculum. However, some thoughts are offered in the following.

As in any curriculum, some subjects are rather dry and tedious. But it would be useful to include fundamental courses such as

- Economics
- Finance (especially managerial accounting and financing)
- Marketing
- R&D Management
- Organizational structure
- Business process modeling and re-engineering
- Intellectual property protection
- Management tools.

For communications and team management, the curriculum should include

- Verbal and written communications
- Psychology, personality types, team dynamics
- Negotiation
- Corporate culture[12]
- Human resources management and legal aspects.

Finally, the program must contain a comprehensive set of software engineering topics, including

- Software engineering economics
- Product management
- Process improvement (the whole CMM body of knowledge)
- Project planning, monitoring, and control
- Requirements management
- Software engineering tools

Again, this is not intended to be an exhaustive list for such a degree, but rather some pointers for future refinement.

7. Summary

Our survey of software companies was intended to surface recurring problems in the software industry, which create major impediments to success. We were surprised to find that although the respondents were predominantly technical managers, the technical challenges were rated lower than the management challenges. Our industry has come a long way. From programming to structured design, to software engineering, we have addressed issues at increasingly higher levels. We are now at a point where we need to seriously address problems at the top of the pyramid, namely the management of the software development process. Process improvement initiatives, as promoted by the SEI, address structural issues at the process and organizational policy level. Other initiatives address the abilities and skills of development staff[11]. This paper suggested ways to improve the skill set of the individuals responsible to orchestrate the development and assembly of software.

The author would like to suggest a sequel to this study. There surely must be excellent software managers in our industry. A statistical survey such as this can only show general tendencies and tends to bury extremes on either side of the spectrum. It would be interesting to identify individual project managers with a proven track record of highly successful projects and to ask them about those key factors they judge to be the basis for their success.

8. Acknowledgements

This study was funded by the Canadian National Research Council and the BC Ministry of Education, Skills, and Technology. Contributions were made by 23 individuals, representing industry, education, and government. We also acknowledge the time spent by our survey participants to reply to a rather lengthy questionnaire. Special thanks go to those at the SPC who designed and conducted the survey. Alice Kloosterboer had the overall responsibility and designed the questionnaire. Yee Chan assisted in the survey design and analysis of the results. Dr. Kal Toth participated in follow-up interviews and prepared the final report.

9. References

(1) De Marco, Tom; Lister, Tim; “Peopleware”, Dorset House, November 1987

(2) Yourdon, Edward; “Decline and Fall of the American Programmer”, Yourdon, 1992

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

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

(6) “Technology and Skills Gap Analysis: BC Software Industry”, Software Productivity Centre, March 1997

(7) Strigel, Wolfgang, B.; “Results from Process Assessments in Software Companies”, PICMET, Portland, OR, July 1997

(8) “Statistical Review of the Software Products Industry (1994)”, Industry Canada, 1995

(9) Strigel, Wolfgang, B.; “Engineering Demographics in the High-Tech Industry”, Simon Fraser University, Nov. 1988

(10) Smith, A.L.,”Competing in the US”, Business Quarterly, Summer 1997

(11) Humphrey, Watts; “A disciplined Approach to Software Development”, Addison-Wesley, 1995

(12) Wiegers, Karl, E.; “Creating a Software Engineering Culture”, Dorset House, 1996

Maximizing Lessons Learned - A Complete Post-Project Review Process

Rick D. Anderson
Senior Software Engineer
Tektronix, Inc.
P.O. Box 500, Beaverton, OR 97077
(503) 627-2630 *rick.d.anderson@tek.com*

Bill Gilmore, Ph.D.
Senior Staff Software Engineer
Intel Corporation
5200 N.E. Elam Young Parkway, JF3-204, Hillsboro, OR 97124-6497
(503) 264-6859 *bill_s_gilmore@ccm.jf.intel.com*

Abstract:

Many organizations hold post-project reviews (also known as post-mortems), but often there is little change as a result and it seems many times a final report is written and then it gets filed in a notebook somewhere, never to be seen again. The real value in holding a post-project review is to improve future projects. This paper presents an easy-to-use six step process for holding post-project reviews, where the primary objective isn't just to hold the review, but to learn from the just completed project and then to use this "wisdom" on future projects. The concept of a Lessons Learned Repository is also presented.

Biographies:

Rick Anderson is currently a Software Quality Leader within the Measurement Business Division at Tektronix. His interests include software process improvement, formal technical review processes and software metrics. Prior to this, Rick spent four years at Interactive Systems, Inc. in Beaverton, Oregon as Manager of Software Development and six years at AG Communication Systems in Phoenix, Arizona as a Software Project Leader. He graduated Summa Cum Laude from Oregon State University with a B.S. in Computer Science and a B.S. in Business Management.

Dr. Bill Gilmore is currently a Senior Software Engineer in the Corporate Quality Group at Intel. Prior to that he was manager for Software Process Improvement at Tektronix, Inc. and has been a consultant for software process and strategic planning. He has worked at the SEI, International Software Systems, BDM and Texas Instruments in the areas of software process, methodology, software project management and software engineering and development. He received a Ph.D. in Astronomy from the University of Maryland and a B.S. in Engineering Physics from Cornell University, and did graduate study in Organizational Development at the University of Texas.

1 Overview

1.1 Introduction

Much insight can be gained by analyzing a software project and identifying what was successful and what was not over the life of the project. **The primary goal of a post-project review (PPR) is to learn lessons from the just completed project that can help planning and engineering on future projects and/or stimulate process improvements.** This learning occurs by raising significant positive and negative issues, identifying causes and suggesting improvements.

While this underlying goal usually stays the same, the specific objectives of the PPR can vary. For example, the objective of the PPR might be to collect issues on specific processes used (e.g., specific process definitions, checklists, and templates) and analyze these to identify and implement related improvements. The objectives of the PPR can also be very specific, such as collecting issues on a perceived project problem. Or the PPR may be to raise issues in general and see what arises. Careful identification of the objectives is important so that the review can be planned to accomplish them.

PPRs empower engineers, allowing them to identify issues and spearhead improvement activities. At a minimum, each project should have at least one post-project review. For larger projects, mid-project reviews (MPRs) might also be helpful, or even reviews at each major milestone.

1.2 Organizational Learning

Many projects hold post-project reviews to capture lessons learned. However, often the lessons learned from the project do not get properly transferred to the next and future projects. This knowledge transfer is an important part of continuous process improvement and on-going organization learning. As figure 1 shows, collecting project data to improve process and quality assets over time is a very important part of on-going organizational learning. Planning a project can be substantially aided by assets such as process definitions, with checklists and templates; a database of project histories of planned vs. actual time and resources used for projects of various complexities; and a repository of lessons learned on previous projects, regarding things to do, risks and hazards to consider, etc. PPRs are a primary mechanism to collect information to build upon and improve such assets.

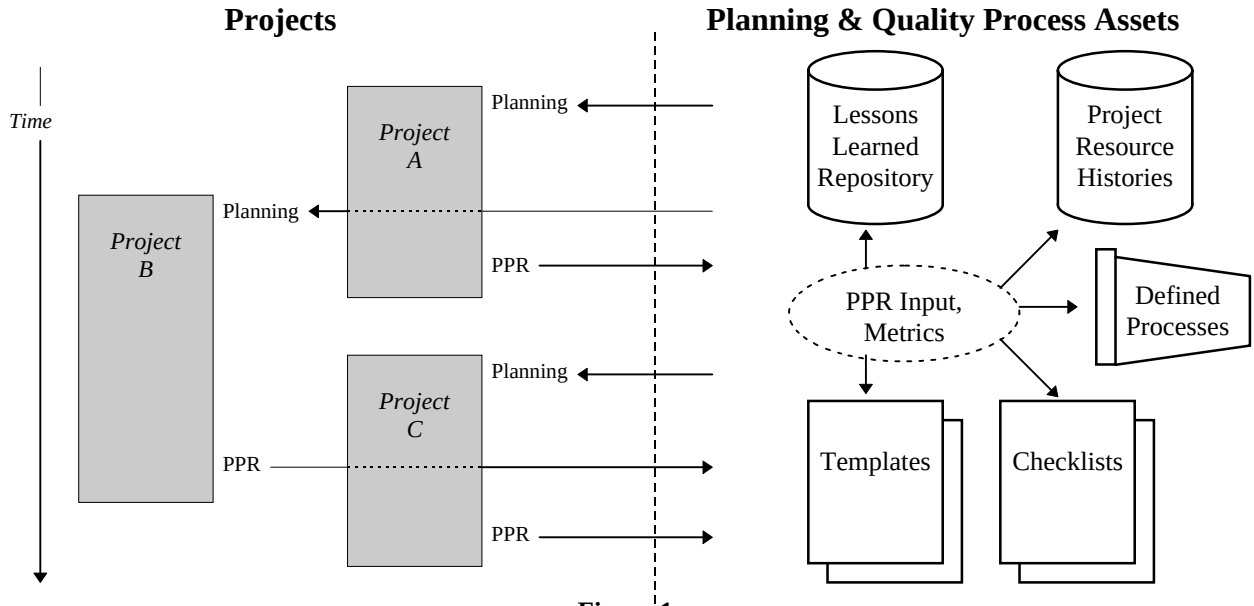


Figure 1

1.3 Lessons Learned Repository

A new concept introduced in this document is that of the Lesson's Learned Repository, or LLR. A common problem with many post-project reviews is that the lessons that were learned from past projects do not get transferred to future projects. Often, a written report is completed, stored in a configuration management system, and never seen again. To address this challenge, a LLR is being created at Tektronix to communicate and track lessons learned. The LLR is a process asset that can, and should, be browsed when planning a new project or anytime software personnel have a question where past project wisdom might apply. The LLR is updated both during the project (as lessons are learned) as well as with every PPR.

The LLR contains information that a planner of a project would like to have available about past projects; it is a database of wisdoms that should aid planning and development on future products. The LLR is organized in such a way to make information retrieval straightforward, minimizing redundancy, and grouping like wisdoms. Information can be extracted by beneficiary, by project phase, by category or keyword (with categories such as requirements, architectures, planning, design, testing, configuration management, inter-functional communication and interaction) or by process improvement activities undertaken.

2 Post-Project Review Process

2.1 Process Overview

Post-project reviews require more than just a meeting of the project team and some meeting minutes in order to be successful and provide value to future software development projects. Figure 2 outlines a six step process and summarizes what activities take place during each step. Each step is then discussed in subsequent sections.

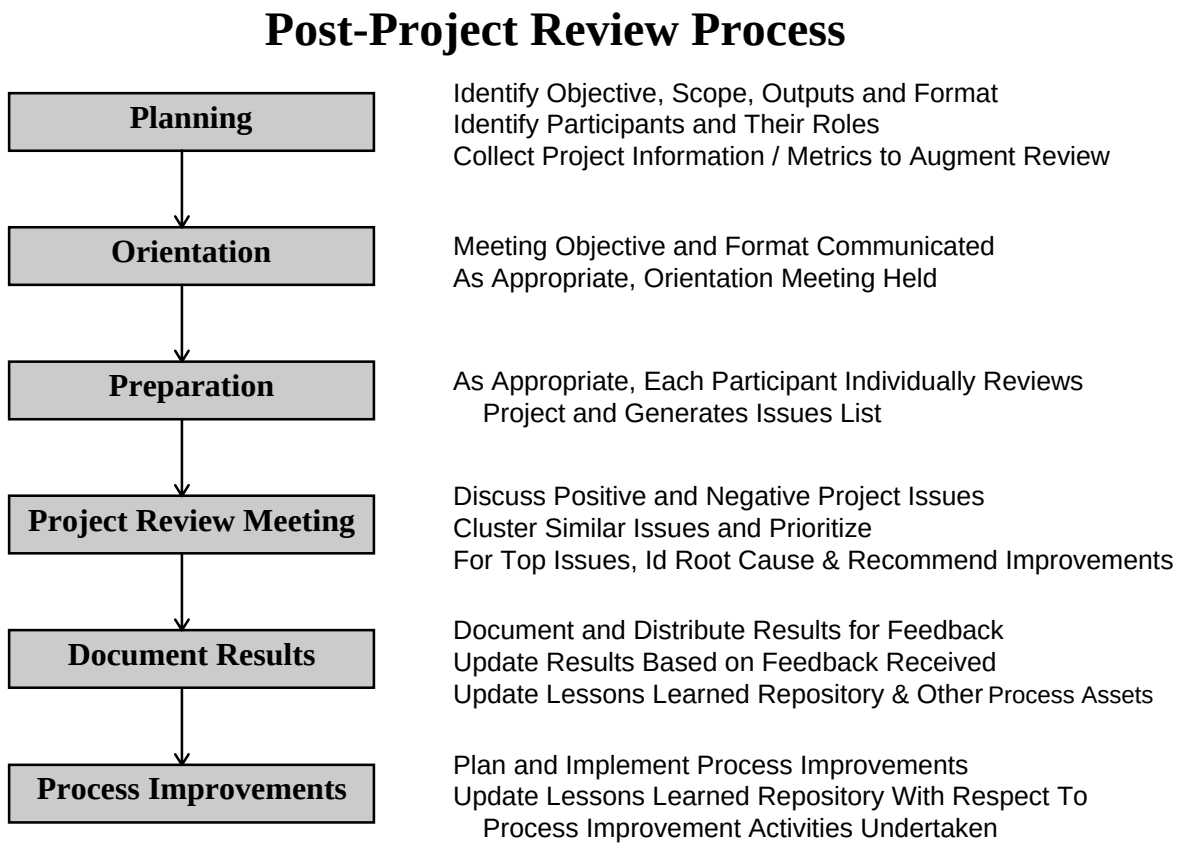


Figure 2

2.2 Planning

The most critical aspect of the planning step is identifying the specific objectives and scope of the PPR. Depending on these, one then plans such things as:

- number and format of meetings
- what outputs you will want (e.g., formal report, issue list, or what?)
- how the follow-up improvements will occur (generally, specifics will depend on improvement needed)
- who will participate, and their responsibilities
- what data might need to be collected

Planning is generally the responsibility of either the Software Functional Manager, the Software Project Leader, or a Software Quality Assurance engineer. The following questions help one identify PPR objectives and develop subsequent plans.

What are the objectives of the review?

- What do you actually want to accomplish at the end of the review? Possibilities include:
  - improve the planning and quality assets (such as checklists, templates and defined processes).
  - motivate and empower the team to improve its own processes.
  - identify focused objectives, like improving communication within the team or among the team and other functional groups.
  - identify the top three things that need improvement and to kick-off teams to implement these improvements.
- Who are the target beneficiaries of the PPR?
  - Is it software management? Upper management? Program management? Hardware? The software group? The next project? It could very well be all of these. This affects whom you include, how you include them, and what you want for outputs of the PPR.
  - By positioning the PPR process as one that the SW Engineers own, control and is primarily for their benefit, the PPR will generally be more successful.

- **What is the scope of the review?**

- Are we reviewing a single project or multiple projects?
- Should the review be a software-only PPR, or one including other functional areas of the project, e.g., hardware, marketing, program management? For multi-function PPRs, separate meetings for each functional group may be needed, followed by an integrated group meeting.
- Are there specific issues you know you want to target or do you want unbiased exploration of the project and open issue identification? Suggesting issues in the orientation step (step 2) pre-biases what issues are raised. Conversely, using emotion graphs (plots of a participants emotions through the project lifecycle; see Appendix C) during the preparation step (step 3) supports open, unbiased issue identification. Depending on the objectives, pre-biasing may or may not be desirable. If you want specific issues addressed, you can use email during the preparation step to identify these prior to the review meeting (step 4), thus reducing meeting time. If you want an unbiased range, it is better to maximize group synergy and allow for more meeting time.
- If you are in doubt about whether to pre-bias the issues, one technique to utilize is to not pre-bias the group, but if the issues that you think are important don't arise after the initial issue generation process, then you can bring them up yourself for discussion.

- **What should the outputs of the PPR be?**

- If information is not captured in a way that is useful to learning and future projects, its value is questionable. The outputs of the PPR should be designed to benefit whomever you identify as the beneficiaries. For example, an executive summary report for software management along with documention of the lessons learned and suggested revisions to process assets (see figure 1) would benefit the software group and aid planning future projects. Note that the conventional concept of a (~15-page) formal document for future reference is probably not the best way to benefit those whom you want to help.

- **What measurements do you want for the PPR?**
  - Plan what data are needed and when they should be presented. Depending on the objectives, you may want to present some data during the orientation step or by email prior to the review meeting step.
  - Typical data might include:
    - Standard project history metrics:
      - list of major milestones with planned versus actual dates
      - the reasons underlying schedule slips, if any occurred
      - number of people as a function of time over the project
    - Number of defects over time, if possible with severity level and type (e.g., requirements, design, ...). You may even want to break this down by product module areas.
    - Amount of time that software tasks were on the critical path (for products with multiple functional areas participating), and when and why.
- **Should issue generation take place prior to the review meeting?**
  - Individual issue generation prior to the review meeting biases the nature of the issues. This is either good or bad, depending upon your objectives.
  - If you are uncertain what to focus on for improvement or what specific issues need to be examined, then save issue generation for the review meeting in order to use group synergy to help identify and formulate the issues.
  - If you have specific topics that you already know you want to analyze, then prior issue generation around these topics, and possibly others suggested by the reviewers, can save time and focus the review meeting more effectively. Issue generation centered around the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) Key Process Areas (KPA's) can also be used (available on the World Wide Web at: *http://www.sei.cmu.edu/technology/cmm.html*).
  - Software group policies, process definitions, checklists and templates should all be reviewed for helpfulness and relevance during the project. Other areas to explore might include: where quality breakdowns occurred, key risks that should be earmarked for future projects, planning difficulties and successes.
  - If issue generation takes place prior to the review meeting, issues can be forwarded to the scribe via email and potentially can be summarized, clustered, and/or prioritized. This technique can be used to reduce group review meeting time commitments. If the scribe doesn't get much email back, you also know that people have not prepared properly.
  - Another pre-review meeting issue generation technique is to utilize an electronic survey. A survey can be prepared and individual opinion on various issues can be measured, hopefully identifying areas of weakness (and thereby prioritizing future issue discussion). This is especially useful when a wide variety of topics need the project team's input or when a PPR objective is to explore specific areas. An example survey can be found at: *http://www.wildfire.com/research/postmortems.html*. The article *A Defined Process For Project Postmortem Review* by Collier, DeMarco and Fearey in IEEE Software, July 1996, is also worth a read.
- **How many meetings should be planned for step 4, the review meeting step?**
  - Obviously, the scope as identified above is a major determinant of the number of meetings.
  - Multiple meetings should be held when either: (1) the objective is to solicit management and engineering issues separately; (2) the PPR is actually a complete project review and multiple functional groups are involved such as hardware, manufacturing, marketing, software, program management and documentation; (3) it is important to separate the surfacing of issues from the analysis of those issues (for example, if time is required to think about what the root cause of the issues might be); or (4) a single meeting would just be too long.
  - Each functional area might have a different view on what went well and what did not. Collecting these separately could provide some useful insight into where views differ and where improvements could be made. Whenever input is collected in separate meetings or with individuals out of the room, the issues should be reported and discussed via a collective feedback session. Then everyone can decide how to deal with the issues.

- **What should the format of the review meeting be?**
  - Typically, the review meeting will have five steps:
    Step 1: surfacing issues
    Step 2: clustering & prioritizing
    Step 3: selecting 3-10 key issues to further analyze
    Step 4: root cause analysis and identifying lessons learned on the key issues
    Step 5: recommending improvements
    A final step might be to summarize the meeting at the end.
  - Be aware that groups brainstorm well, they bring diverse interests and expertise to the table and they arrive at a collective understanding that is better than a single individual can achieve. It is not time efficient for groups to write final drafts, or to define meeting process once the meeting has started.
  - Plan in advance what methods will be used to cluster and prioritize issues, to analyze root causes and lessons learned, and to identify process improvement actions. These steps can be difficult and sometimes lead to wasted time and frustration.
  - The personality, culture, size and diversity of the group will influence the overall format of the review. Getting group buy-in on the process is important and can take place during the orientation meeting (see next section).
  - Consider whether managers should participate in the entire review process or just part of it. Some individuals might be hesitant to provide critical comments while managers are present. Potentially, managers could step out of the room for a specified period of time while comments specific to their processes are gathered anonymously.
- **How much time should be devoted to the review meeting?**
  - This time should be planned in proportion to the size of the project, how much there is to learn and how significant the learning is to future projects. It is also important not to let the entire PPR process drag on over a long period of time.
  - The first three steps (surfacing issues, clustering & prioritizing and selecting top issues) can take anywhere from 1.5 to 5 hours, depending on the issue range (see above), nature of the project and team, and amount of time you decide you can afford. In general, surfacing of issues takes 45-180 minutes, clustering and prioritizing take 15-30 minutes each and selecting top issues takes 15-30 minutes. Use of advanced issue generation and surveys can reduce this group meeting time commitment (but individuals will probably spend equivalent amounts of time on their own).
  - The last two steps (root cause and recommending improvements) require analysis and therefore take time. Root cause analysis can take 1 to 5 hours (often 15 to 90 minutes per issue), and identifying recommended improvements typically cannot be done in less than 30 minutes. Sometime indicated actions for some key issues are straightforward and may need only a little or no analysis and discussion.
  - It can be very useful to have separate meetings for steps 1-3 and steps 4-5, with not more than a week between meetings. This allows for some thinking and reflection between meetings, but keeps the issues warm for analysis.
  - Ensure that enough time has been planned (in the project schedule and/or in the transition to next projects) for steps 5 & 6, documentation of lessons learned and process improvement.
- **Who should participate in the review?**
  - Generally the entire project team should participate in the PPR; however, the objectives of the review determine who should participate. Sometimes only some of the project team should participate.
  - Sometimes the project team will be unaware of its effect on or perception by other groups. Consider getting outside input in addition to that of the project team (e.g., product line management, program management, functional management, other software engineers, hardware engineers, manufacturing engineers, quality assurance, documentation, marketing, mechanical engineering, service, etc.). Getting balanced input will yield better overall results. Careful consideration needs to be given to how best to solicit such input. Potentially several review meetings could be held with different groups of people.

- **What are the various roles for the post-project review process?**
  - At a minimum, the review moderator and scribe roles must be filled. The process leader must assign these roles. The moderator and scribe should be different people, although for a small project, they might be the same person. Training individuals in their role's responsibilities might be necessary.
  - The process leader, review moderator and/or scribe are responsible for preparing the final summary of the post-project review, communicating this to the rest of the organization, entering the lessons learned into the Lessons Learned Repository and entering project history data into the project history database.
  - Some desired attributes of a moderator: (1) understands the functional domain, (2) ability to lead group discussions and mediate disagreements, (3) ability to identify key issues and maintain group focus, and (4) can remain unbiased during review. It should also be stated that moderating can be difficult. Groups tend to wander, go off topic, come up with ideas in waves, stall, and require probing.
  - Some desired attributes of a scribe: (1) good listener, (2) able to extract key points from discussion, (3) able to summarize issues and do so quickly, (4) understands the functional domain.

PPR Roles

Process Leader - responsible for planning the PPR, getting commitment from personnel to participate, selecting the moderator and scribe, and gathering necessary resources. The process leader is usually a software functional manager, software project leader, or SQA engineer.

Functional Management - responsible for committing personnel and resources to support the review activities.

Moderator - responsible for facilitating the review meetings to ensure that they proceed as planned.

Scribe - responsible for recording raw issues, clustered and prioritized issues, identified root causes, and improvement actions. The scribe and/or the moderator also has responsibility to generate and distribute the planned outputs of the review, e.g., a written summary of the review.

Gatekeeper - responsible for watching the clock and making sure one issue does not get discussed for too long and that the review meeting is staying on agenda and time constraints. Role often taken by Moderator.

Reviewers - responsible for objectively reviewing project, providing input on positive and negative lessons learned and suggesting improvement actions. In some cases individual reviewers might be asked to focus on certain areas or processes so as to get more coverage of that area or process.

Software Process Improvement Representative (e.g., from SEPG)- responsible for supporting the PPR; helping with planning, meetings, and follow-on improvements if requested; and over-seeing that the LLR is updated and used.

- **How many project reviews should occur?**
  - Project reviews should be scheduled early in the project during the project planning phase. For small projects, a single PPR is sufficient. For large projects, it can be worthwhile to schedule a project review at the end of every phase. MPRs allow for improvement and needed corrections (e.g., of communication, responsibility, or product definition problems) in the middle of the project, which results in a better end product. In addition, MPRs reinforce the team concept and promote better group communication.
  - Instead of having a long PPR at the end of the project, holding short MPR's at the end of every major milestone can be used. In this case, issue generation usually takes place via email and review meetings are limited to two hours in duration.
- **What other planning questions should be considered?**
  - What do you want to present at orientation, and what kind of orientation should there be (see orientation step)?
  - What do you want reviewers to do during the preparation step (see preparation step)?

- **What is management's role in the PPR process?**
  - Management must support the PPR process. In addition to allocating resources (staff and time) and monitoring the process, they should encourage PPR participation by project members, recognize individuals for active participation and contributions, and be willing to sponsor improvement activities.

2.3 Orientation

The purpose of the orientation step is to:

- communicate the PPR objectives, meeting schedule and timeline, roles and responsibilities, and overall process to the participants;
- ensure the commitment of the reviewers to participate fully.

The process leader coordinates this step; however, the review moderator can also play a role.

- **How will the planning details be communicated?**
  - Orientation can occur either through a written memo, email or by holding a meeting. With memos and email, one hopes to save time through efficiency. However, often there is still some misunderstanding about exactly what is intended to be accomplished. Typically, meetings are more successful for orientation. Communication can be direct and interactive, team members can ask questions about the process and their roles, and the process leader can check the reviewers' commitment. Meeting feedback can be useful to the process leader and moderator and may influence the way the review meetings are held. The orientation meeting can also be used to distribute the review meeting package.
- **What will the review package consist of?**
  - Goal and objectives. A clear statement of the goal (see section 1.1) and objectives (from planning step) of the post-project review.
  - PPR timeline. An overall indication of how long the PPR process will take from start to finish.
  - Meeting notice. This should list meeting location(s), date(s) and time(s), participant list, roles and who to contact if you cannot attend.
  - Agenda. This should list agenda items, responsible parties and duration's.
  - If appropriate, project information and metrics.
  - If appropriate, a checklist of items to consider during issue generation (see Appendix B). If new processes were added as part of this project, checklist items on those specific items might want to be added to the checklist.
  - If appropriate, a specific list of suggestions for what to look at to help generate issues. This could be work-products or memos, or could be one's own experiences if an emotion graph is to be used.
  - A suggested amount of time (range) for individuals to spend doing preparation.

Note: The review package should be distributed at least five days prior to the meeting if individual issue generation is being used, in order to give people time to generate a list of issues.

2.4 Preparation

This step refers to individual preparation prior to the review meeting. Depending on the PPR objectives and format of the review, this step might include completion of a survey or email input on the issues. Alternatively, if it is not important to the objectives, no individual preparation may take place prior to the review meeting. The details of exactly what should happen during this step vary by PPR and should be outlined to participants during the orientation step. Assuming individual issue generation will take place, the following questions are often asked:

- **What materials should individuals review?**
  - Project work-products, personal notes, status reports, schedules, memos, project history data and other materials generated during the project timeframe should be considered.
  - Consider all materials in the review package, including participant checklists and project information.

- Emotion graphs (described in Appendix C) can be useful for identifying issues.
- New processes added during project. Were they effective and beneficial?

- **Are there important hints for individuals during this preparation step?**
  - To maximize effectiveness, set aside a time just for PPR preparation and spend it in an undisturbed environment.
  - The amount of time to spend will vary by individual and project. In general, individuals should spend enough time to satisfy themselves that all key issues have been found (brainstorm until no more issues can be identified).
  - Ask the question, *"What have I learned from this project that I'd like to pass on? "*
  - If in doubt as to relevance, include the issue. Do not restrict issues by assuming that a given topic is "outside the domain of relevance". For example, a continually broken copier may turn out to be both relevant and actionable.
  - Identify critical issues versus unimportant ones. Go through the entire brainstormed list and identify the key issues.
  - Distinguish symptoms of problems from the real causes.
  - Identify both positive and negative issues.
  - Don't wait until the last minute to prepare. Generate issues and then let them sit overnight. Often additional issues will be remembered during this "settling" time.

2.5 Review Meeting

The PPR meetings are directed by the review moderator. **The meeting format will vary for each PPR and is dependent on the previously determined objectives.** These objectives will determine what activities take place in the meeting, in what order and how many meetings will occur. Explicitly taking the time to plan the review meeting is critical to make this step a success.

As mentioned in the planning section, the meeting phase of the PPR generally consists of the following activities:
- surface issues
- cluster and prioritize issues
- select top 3-10 issues
- analyze root causes and identify lessons learned for key issues
- identify and recommend improvement actions

All of these activities might not occur in a single meeting. The following questions should be answered with respect to the review meeting:

- **Setting Expectations**
  - Each meeting should begin by clearly stating the objectives and agenda of the meeting (5 minutes). Also be clear about the overall process and the end goal.
  - As the review moves from step to step, clearly explain to the participants that this is happening. This will cause the team to focus on the task at hand.
  - The last step of the final review meeting should be to summarize. This helps deepen commitment of the group, reaffirms the consensus and helps influence future change.

- **Surfacing Issues**
  - How you chose to surface issues is a key factor, i.e., in advance of the meeting or during the meeting. This determines how much brainstorming you want to plan versus the amount of time to discuss and flush out more details of various issues. It can be helpful to discuss what worked well first, and then follow with what didn't work well. Starting with the positive sets an open tone.

- Many different techniques for surfacing issues during a review meeting exist. The moderator should try to ensure that everyone participates; using the round table approach is often a good solution. Here, each member is asked to raise one issue. The moderator keeps going around the table until all issues are raised.
- While identifying issues, the scribe should document all issues, both positive and negative. A typical PPR might identify 50 to 200 issues.

- **Clustering and Prioritizing**
  - The order for doing prioritization and clustering can be reversed. Most often grouping takes place then prioritization; however, in some cases, additional learning can be accomplished by prioritizing first and then grouping. By prioritizing first, connections and root causes can become apparent that otherwise may not have been seen.
  - When clustering like issues together, the particular categories used will vary based on the issues documented. Experience shows that it is useful to use a combination of CMM categories, categories related to your objectives and categories that are meaningful to the group.
  - The prioritization can be done by individual vote (each person votes for their top three or five issues; each has 25 points to assign anyway they see fit, etc.), triple ranking (like individual voting, but voting happens in three rounds, with the lowest vote-getters discarded each time), discussion and group consensus, or some other method. Issues can be prioritized from 1..n, or as high/medium/low.
  - The moderator must be very clear about the criteria to use for prioritization. Failure to do this might result in different people using different criteria. An example prioritization criteria might be: the group should ask themselves which of the issues are most critical to improve on, to institutionalize or to pass on to the next project.

- **Selecting Top Issues**
  - Which key issues you want to analyze and the depth and degree to which you want the group to do analysis and recommendations must be decided. Analyzing all key issues is very time consuming and usually unnecessary. Sometimes it is best for the process leader to select which issues will be analyzed. Recognize, however, that participants might feel that certain high priority issues are being skipped for political or other reasons. The moderator should be up-front about why select top issues are not being discussed further.
  - In some cases, it is appropriate to not address one of the top issues, for example, if the group cannot bring change to the issue, or the appropriate people are not in the room to discuss it, or if process improvement efforts are already underway.
  - In some instances it is appropriate to take certain top issues and move them outside of the PPR process via a small team. In these cases, the issue can be assigned to the team, the team discusses the root causes and suggest improvements, and then they report back to the PPR participants with the results and possible further discussion.

- **Root Cause**
  - For the top issues selected, focus on identifying what the underlying cause of a success or problem was (the root cause). Try to distinguish symptoms from real causes. For example, lack of training might not have been raised as an issue, but issues such as missed deliverable dates, high defect rates and low defect finding rates in formal technical reviews could all be caused by inadequate training. Cause and Effect Diagrams and other textbook methods can be used here effectively.
  - A key question to ask*: "What have you learned about <issue> that you think is important to pass on to people on future projects?"* Do not discuss history, e.g., "this went good" or "that went bad". This history is worth little to future projects. The meat is in "why". To surface this, stimulate thinking about root cause, ask what was learned (*"What is it you know now that you didn't know at the beginning of the project?"*)
  - The root cause can be rephrased as a lesson learned. For example, if the issue is lack of training, the lesson learned might be that future projects' staff should be adequately trained at the start of the project on processes and tools used during the project.

- **Recommending Improvements**
  - The final activity during the review meeting is to take the top three or ten issues that you did root cause analysis on and to recommend process improvement actions that should take place. The purpose of this is to plant the initial seeds for the improvement project. Most often if you know what went wrong, you also have a good idea on how to correct and improve it. For each of the high priority lessons learned, try to identify one or more specific activities that will positively reinforce or improve the lesson.
  - Try to separate root cause analysis and recommended improvements. If solutions are raised during root cause analysis, simply document them in another color or on another flipchart. In any case, be sure to spend some time brainstorming on just suggested improvements.
- **What are some key points to consider as a moderator and scribe?**
  - Good recording is important:
    - Listen closely to capture the issues being stated. Don't transcribe the words; abstract and capture issues.
    - At times, use judgment in allowing discussion in order to enable real issues to be discovered and surfaced. At other times, issues will come fast and furious; try to record things "true" to the way they are surfaced or intended.
    - The person raising the issue should own the wording. Ensure that this happens. Use of flipcharts or overhead projection machines is a good technique, as participants can see the comments as they are documented.
    - If possible, examples given should also be succinctly documented.
    - Number the issues, for later grouping.
    - A summary table with columns for issues, priority, root cause and improvement actions can be used.
  - Stay focused on the step at hand, e.g., if you are surfacing issues, avoid prioritization. Allow analysis if it is likely to lead to further surfacing or clarification of issues; this requires moderator judgment.

2.6 Document Results

The documentation stemming from a PPR should take whatever form(s) best support planning and engineering on future projects and stimulate process improvements. The final PPR documentation could consist of a list of issues targeted for improvements, updates to a LLR, updates to the project history database, and a summary report to management.

Possible audiences for PPR documentation are: people on subsequent, related projects; others in the software group; other functional groups; management; and people doing process improvement in any of these areas.

- **What specific form should PPR documentation have?**
  - The objectives of the PPR will dictate the documentation form. **The single most important point is that any documentation produced should be specifically targeted at the beneficiaries identified during the planning step.** Writing a formal report just to document issues is not good practice.
- **Is there any intermediate documentation that is different than the final documentation?**
  - Yes. Prior to a review meeting, the process leader or moderator might email a preliminary list of group generated issues (often called the topic list). After the surfacing of issues and following the selection of the top three to ten key issues, the scribe might release an organized list of the issues generated thus far (often called the preliminary findings). Following root cause analysis and recommendations, a list of improvement actions along with text describing lessons learned and the results of root cause analyses might be disseminated to the PPR reviewers for inspection and editting (often called the summary findings). In addition, preparation of documentation into final forms may occur for intended future beneficiaries, e.g., updates to the LLR and project history databases; updates to process definitions, templates, and checklists; and summary reports to management (often called the final summary report).
- **What is the documentation process?**
  - The scribe or moderator is responsible for intermediate documentation. The moderator, scribe or process leader has responsibility for the final documentation forms, including process asset updates and final report.

- Results, both intermediate and final, should be written down, reviewed, and reworked until agreed upon by the PPR participants. Participants need to own the results and understand them, in order to directly learn from them and to support corrective actions based on them.

- **What might a final summary report look like?**
  - A formal written report should be well organized with increasing depth. The report might contain the following sections:
    - Project summary (limit to one paragraph);
    - Summary of top three to ten lessons learned and recommended improvements;
    - Full list of issues in clustered or prioritized order;
    - Project details (major milestones, project staffing levels, software staffing levels; planned vs. actual).

- **How should results be communicated?**
  - Results of the PPR should be formally communicated to the intended audience (decided during the planning step). Wide-spread communication is important so that others learn from your lessons and pick-up your proposed improvement actions.
  - One method for communicating results is to have a joint team meeting with the people from the old project and those from the new one. By personally sharing lessons learned in this fashion, they are more apt to take them to heart.
  - The formal report should be placed into the configuration management system for documents and possibly posted in some highly viewable place for visibility.
  - The Lessons Learned Repository and project history database should be updated with PPR results. This will enable better planning of future projects.

2.7 Process Improvement

The final step is to implement process improvements so as to further implement, refine and reinforce those aspects of the project that were successful and to change the behavior for those aspects of the project that were unsuccessful. The PPR process is not complete unless process improvement activities are launched.

There are many different ways to pursue improvement. In general a *plan-do-check-act* system is used. Refer to your specific software group policies or various text books available on how to implement process improvements.

As process improvement activities are undertaken, the Lessons Learned Repository should be updated to reflect and track this fact. This will allow projects to share not only lessons learned, but also what process improvement activities were successfully or unsuccessfully implemented.

3 Summary

Key features of the post-project review process described herein include:

- an easy-to-use six step process to plan and carry out a PPR based-on objectives for improvement and organization learning;
- connection of PPRs with improving process assets used on future projets, e.g., histories for planning, lessons learned, and defined processes including templates and checklists;
- explicit focus on using the results of a PPR to benefit future projects;
- introduction of the concept of a Lessons Learned Repository to cumulatively (project by project) build a base of organizational learnings and wisdoms regarding the organization's specific software engineering challenges. The LLR also makes this knowledge easily accessible to software staff and managers, and it is useful for every new software project.

The PPR process described was established as a result of reviewing successes and failures in conducting and using PPRs over several years. In the last year of using this process, we have observed and gotten participant feedback as to the following outcomes:

- PPRs are better planned, organized and generate more useful findings;
- PPRs are easier to plan, set-up, lead and organize subsequent improvement for, because the process is now defined;
- participants feel positive about their use of time for the PPR and feel empowered to improve the processes that strongly affect their daily work lives;
- more process improvement activities are launched and conducted by software engineers;
- participants are energized for their next project, with optimism that some things will be improved.

We caution that these observations are qualitative. Explicit measurement of the PPR process to identify benefits and potential improvements has not yet been established. Undoubtedly the process will evolve and be improved. Nonetheless, the current process is now a basic practice for us; it is defined, trainable and transferable; and it serves as a useful cornerstone for our software process improvement activities.

Appendix B - Participant Brainstorming Checklist

The following checklist can be used by the PPR participants during the individual preparation phase of the PPR process to help stimulate issues. This is not a complete list!

- Where did process or quality breakdowns occur in the project?
- Were there key risks which were not identified during the project?
- What positive aspects of the project should be earmarked for future projects?
- Where could planning of been better?
- Where was documentation good? Where was documentation bad?
- What checklists were used? How should they be improved?
- What documentation templates were used? How should they be improved?
- Things to review:
    - successful and unsuccessful schedule performance
    - cost performance
    - technical performance
    - managerial performance
    - support tools and equipment, including configuration management
    - training and skills adequacy
    - testing and defects
    - requirements
    - project planning
    - architecture
    - coding
    - software quality assurance
    - hardware
    - team and project organization structure
    - team and project communication and coordination
    - technical reviews and inspections
    - customer issues
    - product documentation
    - manufacturing
    - marketing
    - operations
    - subcontractor relationships
    - new processes improvements added during project
- Review personal notes, status reports, old email, etc.
- Plot emotion level through project lifecycle; identify why emotion levels were high and low and what caused changes in direction.
- Helpful Hints:
    - Identify both positive and negative issues
    - Be objective
    - Raise issues; don’t resolve them
    - Focus on issues, not on people

Appendix C - Emotion Graphs

Emotion graphs are plots of a participant's emotions through the project lifecycle. By examining the high and low points and the reasons underlying these, useful insight can be gained into processes and lessons learned. The vertical axis is the participant's emotion level (low to high) and the horizontal axis is the project timeline from project concept to ship. The participant's emotion level, with highs, lows, etc., is plotted on the graph. Significant events in the project can be noted directly on the graph. Separately, the participant identifies the underlying causes of the emotions and the reasons underlying the emotions. Most of the causes and reasons can be directly translated into issues for process improvement and/or lessons learned.

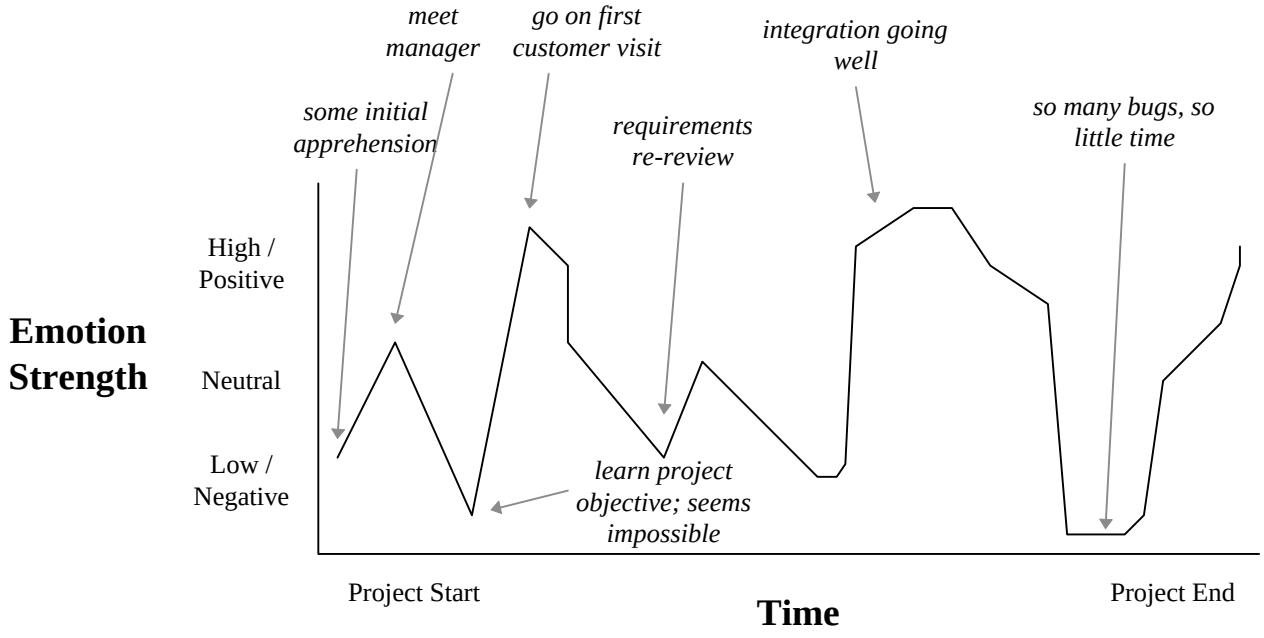


Figure 3

A Survey of Base Process Activities Towards Software Management Process Excellence

Y. Wang, I. Court, M. Ross, G. Staples, G. King and A. Dorling †

Research Centre for Systems Engineering
Southampton Institute, Southampton SO14 0YN, UK
Tel: +44 1703 319773, Fax: +44 1703 334441
Email: wang_s@solent.ac.uk or yingxu.wang@comlab.oxford.ac.uk

† Centre for Software Engineering, IVF, Sweden
dorling@qai.u-net.com

Abstract: A survey has been designed to seek the practical foundation of base process activities (BPAs) in the software industry and to support the research in modelling the software processes. A superset of BPAs [1] compatible with the current software process assessment (SPA) models, such as the SPICE, CMM, ISO 9000 and BOOTSTRAP [2-3], were identified for construction of the questionnaires.

This paper reports the survey findings on BPAs in software management processes. A summary of the current software management process techniques and practices modelled by 170 BPAs in 18 processes and five categories. Each BPA is benchmarked on the attributes of mean importance and ratios of significance, practice and effectiveness.

Based on the benchmarks, and by comparing with the current practice of the reader's organization, recommendations can be given on which specific areas need to have processes established first, and which areas should be highest priority for process improvement.

Key words: Software engineering, software process, base process activities, management process, survey, benchmark, SPA, SPI, SPRM

1. Introduction

This work is based on a large-scale worldwide survey of base process activities (BPAs) towards software process excellence [4-5]. The serial survey is conducted by the Research Centre for Systems Engineering at Southampton Institute UK, in collaboration with the IVF Centre for Software Engineering, European Software Institute and BCS QSig. The survey has been carried out not only on Internet via the professional lists such as the SPICE (SUGAR), WWSPIN, ISO 9000, ISO9000-3, and AMI, but also at a number of international conferences on software engineering and software processes, such as the ICSQ'96 (Ottawa, Canada), SP'96 (a congress incorporated ICSP'96, SPICE'96, SPI'96 and ISCN'96 in Brighton, UK), and SQM'97 (Bath, UK), etc.

This paper reports the benchmarks of software management processes. The paper contains detailed and quantitative evaluation of a set of 170 BPAs in 18 management processes and five categories. For each BPA, the mean weighted importance in process and the ratios of significance, practice and effectiveness are benchmarked. Characteristic curves of each management process are derived based on the data. Other parts of the serial survey on organisation processes and software development processes are reported in [4-8].

Based on the benchmark data, software process practices in a software development organisation (SDO) can be diagnosed and evaluated quantitatively. Process improvement opportunities can be identified and prioritized based on the significance and effectiveness of the BPAs within the SDO.

2. Design of the questionnaire on software management processes

2.1 Model of the software management processes

In comparative analysis of the current SPA models, such as the SPICE, CMM, ISO 9000 and BOOTSTRAP, it is found the structural factors which rule software processes are organisation, development and management. In a software development organisation (SDO), the software development processes and the management processes are parallel counterparts. The former is the producer of software products; the latter is the supporter and controller of time, resources and quality. Both parallel processes are performed in a common environment of the organisation processes.

Based on this view, a software process reference model (SPRM) [1] has been developed with a hierarchical structure consisting of three subsystems, 12 process categories, 55 processes and 444 BPAs. The management process subsystem based on the SPRM is modelled as shown in Table 1.

Table 1. Structure of the management processes

Category No.	Process category	Process	BPA
1	Project planning	4	45
1.1		Project plan	20
1.2		Project estimation	7
1.3		Project risk avoidance	11
1.4		Project quality plan	7
2	Project management	6	55
2.1		Process management	8
2.2		Process tracking	15
2.3		Configuration management	8
2.4		Change control	9
2.5		Process review	8
2.6		Intergroup coordination	7
3	Contract and requirement management	4	42
3.1		Requirement management	12
3.2		Contract management	7
3.3		Subcontractor management	14

3.4		Purchasing management	9
4	Document management	2	17
4.1		Documentation	11
4.2		Process database/library	6
5	Human resource management	2	11
5.1		Staff selection and allocation	4
5.2		Training	7
Total	**5**	**18**	**170**

2.2 Sample space of the SPRM

Because of the partial overlaps in the current SPA models as shown in Fig.1, we need to define a superset of BPAs identified in all models. The sample space defined in the SPRM consists of 444 BPAs which is a superset of those identified in the SPICE, CMM, ISO 9000 and BOOTSTRAP. The BPAs are equivalently known as the 201 base practices (BPs) in the SPICE [9-11]; the 150 key practices (KPs) in the CMM [12-14], the 177 management issues (MIs) in the ISO 9000 [15-17], and the 201 quality system attributes (QSAs) in the BOOTSTRAP [18-20].

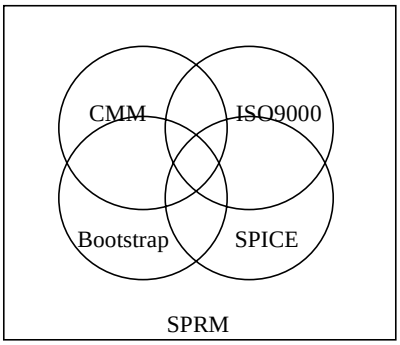
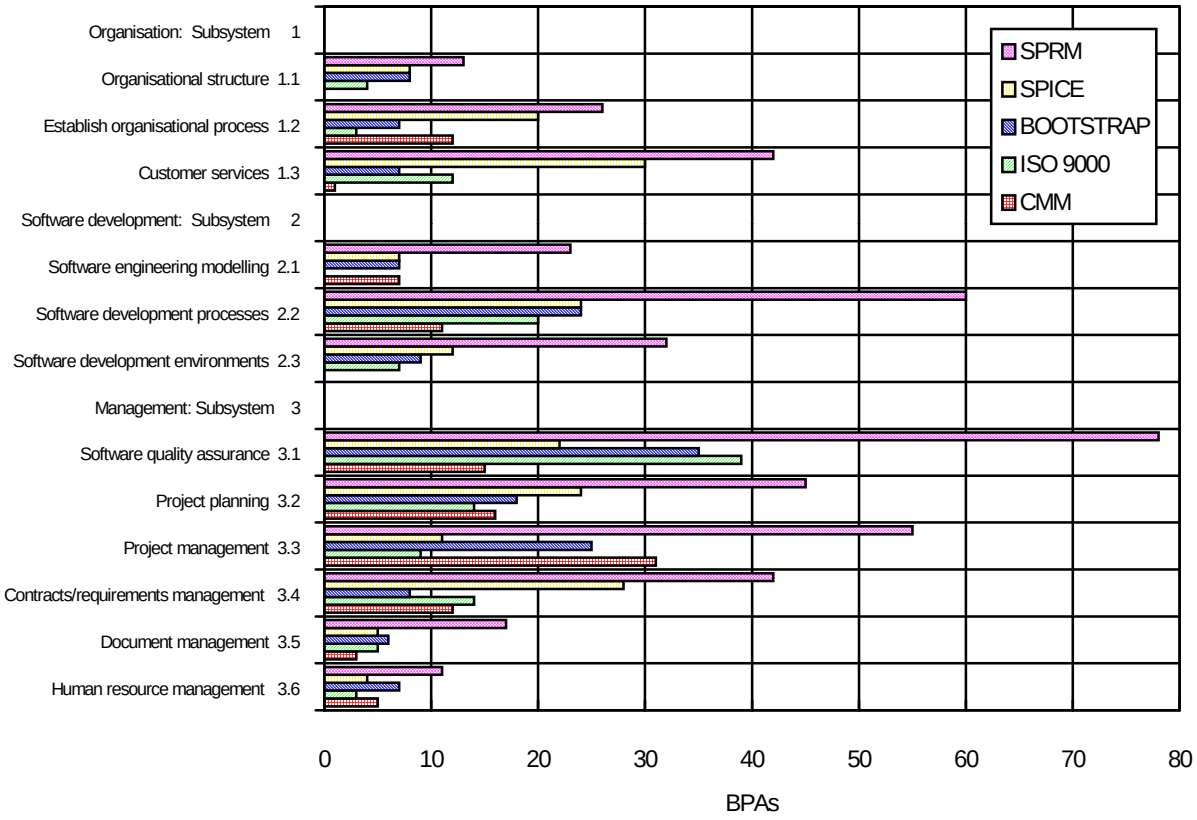


Fig.1 Sample spaces of the current SPA/SPI models

With regard to the SPRM reference model, an overall profile of the CMM, ISO 9000, BOOTSTRAP and SPICE, as shown in Fig.2, can be derived based on the analyses [1-3]. The figure provides an objective view of the emphases, orientation and inter-relationship of the five models in three subsystems and 12 process categories. For detailed mapping between the current models, see [1-3].

2.3 Multi-attribute questionnaire design

The traditional questionnaires of the existing SPA models are one-bit yes/no checklist. Some recent models adopt two-bit four-level adequacy ratings such as fully, largely, partially and not existence of the practices. This survey develops a set of multi-attribute questionnaires, which adopts enhanced three-bit questions, enabling information for each BPA to be collected in a combined domain determined by a production of the attributes of significance, practice and effectiveness in application.



Note: The legend shows the sequence of the models from top-down in each process category

Fig.2 Profiles of the current SPA models

The survey questionnaire listed the BPAs and asked an organization to give an importance weighting for each BPA (on a scale of 0 to 5), to state whether or not they used the BPA in practice, and whether or not they thought it was effective. Based on the raw data, the survey benchmarks for each BPA on the mean weighted importance and the ratios of significance, practice and effectiveness are derived as statistical references for software process establishment, assessment and improvement.

3. Data statistics and processing methods

This section introduces the data statistic and processing methods for the survey. For each BPA, the mean weighted importance and the ratios of significance, practice and effectiveness in application are defined by the following set of formulae.

3.1 Mean weighted importance of the BPAs

The domain of the weights for importance of a BPA, w_i, is defined in [0,1, ..., 5]. The statistical mean weighted importance of a BPA is defined as a mathematical average of the weighted values in the samples as follows:

$$W = \frac{\sum_{i=0}^{5} w_i * i}{n_W} \quad (1)$$

where, n_W is the total number of samples in the survey.

3.2 Distributed values of the BPAs in practice

3.2.1 Ratio of significance for a BPA

The importance weighted for a BPA, w_i, is defined within [0 ... 5]. The numbers of heavy ($3 \le w_i \le 5$) and light ($0 \le w_i \le 2$) weights for a BPA's importance in the total samples n_W, n_w and $n_{\bar{w}}$, are categorised by:

$$\begin{aligned} n_w &= \#\{\, w_i \mid w_i \ge 3 \wedge w_i \le 5 \,\} \\ &= \sum_{i=1}^{n_W} \{\, 1 \mid w_i \ge 3 \wedge w_i \le 5 \,\} \end{aligned} \quad (2)$$

and

$$\begin{aligned} n_{\bar{w}} &= \#\{\, w_i \mid w_i \ge 0 \wedge w_i \le 2 \,\} \\ &= \sum_{i=1}^{n_W} \{\, 1 \mid w_i \ge 0 \wedge w_i \le 2 \,\} \end{aligned} \quad (3)$$

respectively.

Based on the above definitions, the ratio of significance of a BPA, r_w, which has been heavily weighted in the survey, can be defined as:

$$r_w = \frac{n_w}{n_w + n_{\bar{w}}} * 100\% \quad (4)$$

and similarly, the ratio of non-significance of a BPA, $r_{\bar{w}}$, is defined by:

$$\begin{aligned} r_{\bar{w}} &= \frac{n_{\bar{w}}}{n_w + n_{\bar{w}}} * 100\% \\ &= 1 - r_w \end{aligned} \quad (5)$$

3.2.2 Ratio of practice for a BPA

Assume that n_p and $n_{\bar{p}}$ are numbers of the practised and non-practised answers for a BPA in the survey respectively, the ratio of practice of a BPA, r_p, is defined as:

$$r_p = \frac{n_p}{n_p + n_{\bar{p}}} * 100\% \quad (6)$$

and the ratio of non-practice of a BPA, $r_{\bar{p}}$, is the rest part:

$$r_{\bar{p}} = \frac{n_{\bar{p}}}{n_p + n_{\bar{p}}} * 100\%$$
$$= 1 - r_p \quad (7)$$

3.2.3 Ratio of effective for a BPA

Assume that n_e and $n_{\bar{e}}$ are the numbers of the effective and non-effective answers for a BPA in the survey respectively, the ratio of effectiveness of a BPA, r_e, is defined as:

$$r_e = \frac{n_e}{n_e + n_{\bar{e}}} * 100\% \quad (8)$$

and the ratio of non-effectiveness of a BPAs, $r_{\bar{e}}$, is the rest part:

$$r_{\bar{e}} = \frac{n_{\bar{e}}}{n_e + n_{\bar{e}}} * 100\%$$
$$= 1 - r_e \quad (9)$$

3.3 Characteristic value for a BPA

The practical characteristics of the BPAs in the management processes can be combinatorially represented by three attributes: the ratios of significance (r_w), of practice (r_p), and of effectiveness (r_e). A characteristic value, φ, is introduced in Formula (10) which is a production of these three percentages:

$$\varphi = [(r_w*100) * (r_p*100) * (r_e*100)] * 100\% \quad (10)$$

φ gives a combined indication of the BPA's significance, practice and effectiveness. The higher the value of φ, the more important and effective the BPA in practice; and vice versa. Therefore φ can be used to index the importance and effectiveness of a BPA in practice.

4. Benchmark of project planning processes

This section provides statistical data of the survey for 45 BPAs in the four project planning processes. Based on the benchmark, characteristic curves of the project plan, project estimation, project risk avoidance and project quality plan processes are derived.

4.1 Benchmarked findings on project planning processes

This subsection reports the survey findings on the project planning processes. Benchmarked results on 45 BPAs in four processes are listed in Table 2. Table 2 shows, for each BPA, the

mean importance weighting (W), the percentage of organizations rating the BPA highly (i.e. weighting ≥ 3) significant (r_w), the percentage of organizations that used the BPA (r_p), and the percentage that rated it as effective (r_e). The final column φ, the characteristic value, can be used to index the importance and effectiveness of a BPA in practice. These attributes are derived according to Formulae 1, 4, 6, 8 and 10 as defined in Section 3 respectively (the same applies in the following sections). In Table 2, the reference number is the serial number of questions in the survey and the SPRM model [1, 4-5].

Table 2. Benchmark of project planning processes

Ref. No.	No.	BPAs	W [0 .. 5]	r_w (%)	r_p (%)	r_e (%)	φ (%)
	1	General project plan					
275	1.1	Assign project proposal team	3.7	100	92.9	92.3	85.7
276	1.2	Design project process structure	3.9	100	78.6	92.3	72.5
277	1.3	Determine reuse strategy	3.4	71.4	35.7	84.6	21.6
278	1.4	Establish project schedule	4.5	100	100	93.3	93.3
279	1.5	Establish project commitments	4.1	100	92.9	92.9	86.2
280	1.6	Document project plans	4.1	93.8	93.8	93.3	82.0
281	1.7	Conduct progress management-reviews	3.9	100	92.9	92.3	85.7
282	1.8	Conduct progress technical-reviews	3.6	92.9	76.9	84.6	60.4
283	1.9	Management commitments in planning	3.9	93.3	71.4	83.3	55.6
284	1.10	Determine release strategy	3.4	73.3	73.3	84.6	45.5
285	1.11	Plan change control	3.4	78.6	53.8	91.7	38.8
286	1.12	Defined plan change procedure	3.1	71.4	53.8	91.7	35.3
287	1.13	Plan development	4.1	100	100	92.9	92.9
288	1.14	Plan testing	4.0	92.9	92.3	84.6	72.5
289	1.15	Plan system integration	3.9	92.9	83.3	100	77.4
290	1.16	Plan process management	3.6	85.7	91.7	91.7	72.0
291	1.17	Plan maintenance	3.6	93.3	78.6	85.7	62.9
292	1.18	Plan review and authorisation	3.4	85.7	61.5	83.3	44.0
293	1.19	Assign development task	3.6	78.6	92.3	92.3	66.9
294	1.20	Adopt project/process planning tools	2.9	64.3	57.1	84.6	31.1
	2	Project estimation					
295	2.1	Estimate project costs	3.8	87.5	92.9	84.6	68.8
296	2.2	Estimate project time	4.4	100	100	92.9	92.9
297	2.3	Estimate resources requirement	4.5	100	100	73.3	73.3
298	2.4	Estimate staff requirement	4.3	92.9	100	83.3	77.4
299	2.5	Estimate software size	3.9	86.7	69.2	81.8	49.1
300	2.6	Estimate software complexity	3.4	78.6	41.7	90.9	29.8
301	2.7	Estimate critical resources	3.8	86.7	46.2	91.7	36.7
	3	Project risk avoidance					
302	3.1	Identify project risks	3.8	88.2	50.0	86.7	38.2
303	3.2	Establish risk management scope	3.3	78.6	30.8	66.7	16.1
304	3.3	Identify unstable spec. related risks	3.3	81.3	43.8	69.2	24.6
305	3.4	Identify process change related risks	3.1	73.3	28.6	63.6	13.3
306	3.5	Identify market related risks	3.8	93.3	64.3	83.3	50.0
307	3.6	Analyse and prioritise risks	3.4	73.3	40.0	71.4	21.0

308	3.7	Develop mitigation strategies	3.1	73.3	40.0	58.3	17.1
309	3.8	Define risk metrics for probability/impact	2.9	75.0	20.0	61.5	9.2
310	3.9	Implement mitigation strategies	3.1	78.6	28.6	63.6	14.3
311	3.10	Assess risk mitigation activities	2.9	68.6	33.3	58.3	13.4
312	3.11	Take corrective actions for identified risk	4.0	93.3	73.3	73.3	50.2
	4	Project quality plan					
313	4.1	Plan SQA	4.1	94.1	88.2	88.2	73.3
314	4.2	Establish quality goals	4.2	100	80.0	86.7	69.3
315	4.3	Define quality quantitative metrics	3.9	88.2	75.0	81.3	53.8
316	4.4	Identify quality activities	4.1	100	78.6	85.7	67.3
317	4.5	Track project quality goals	3.8	94.1	76.5	81.3	58.5
318	4.6	SQA team participate in project planning	3.6	86.7	57.1	84.6	41.9
319	4.7	Plan maintenance	3.3	73.3	71.4	85.7	44.9

4.2 General project plan process

The characteristic curves of the general project plan process are derived in Fig.3. In Fig.3, the mean weighted importance (W) scaled 0~5 are multiplied by 10 for plotting in a suitable scale with the other variables (the same applies in the following figures).

All the BPAs in this process are weighted important, practical and effective, except BPA 1.3 which lacks practice currently. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as BPAs 1.1, 1.4-1.7, and 1.13. For process improvement, the priority can be put on the BPAs which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w). For example, the BPAs 1.2, 1.3, 1.11 and 1.18.

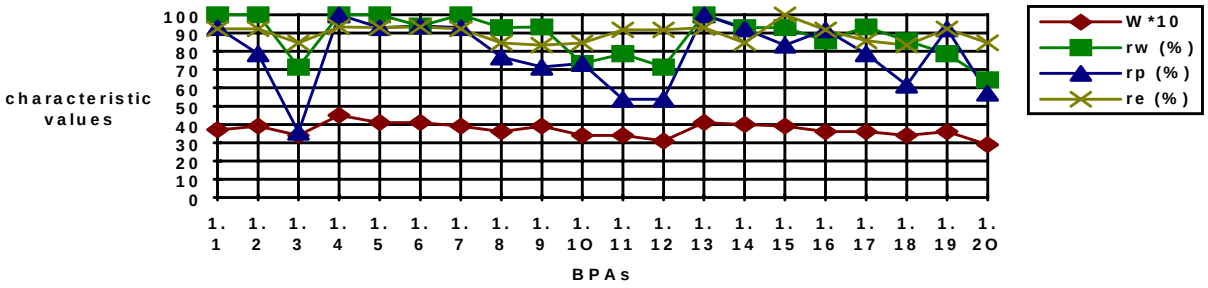


Fig.3 Characteristic curves of the general plan process

4.3 Project estimation process

The characteristic curves of the project estimation process are derived in Fig.4. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as BPA 2.1, 2.2 and 2.4. For process improvement , the priority can be put on the BPAs 2.6 and 2.7 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

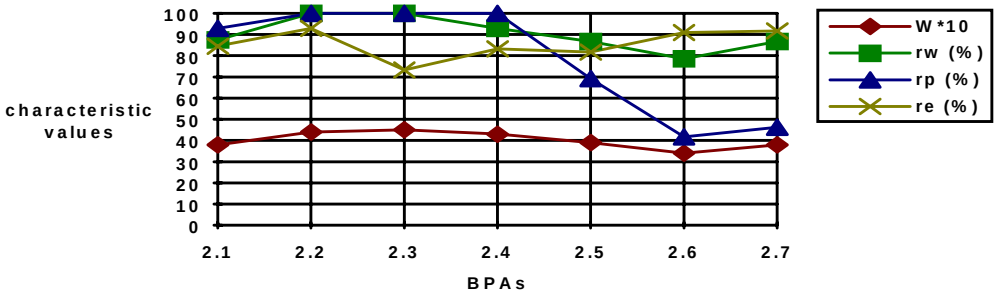


Fig. 4 Characteristic curves of the project estimation process

4.4 Project risk avoidance process

The characteristic curves of the project risk avoidance process are derived in Fig.5. This process is not heavily weighted and practised according to the data. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 3.5 and 3.11. In this process, almost all the practices of the BPAs need to be improved because of lacking practices.

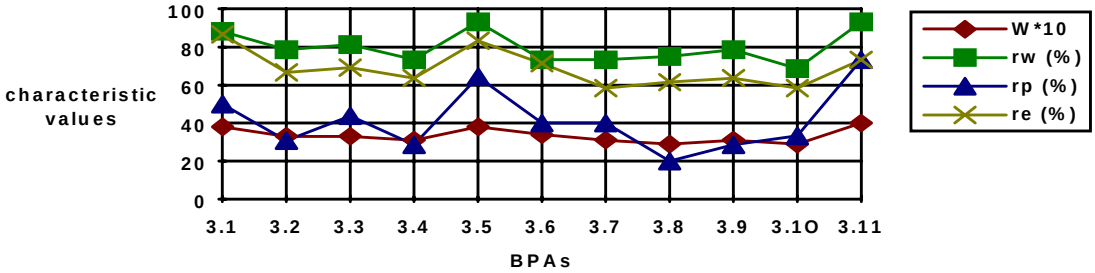


Fig. 5 Characteristic curves of the project risk avoidance process

4.5 Project quality plan process

The characteristic curves of the project quality plan process are derived in Fig.6. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 4.1, 4.2 and 4.4. For process improvement, the priority can be put on the BPAs 4.2, 4.4 and 4.6 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

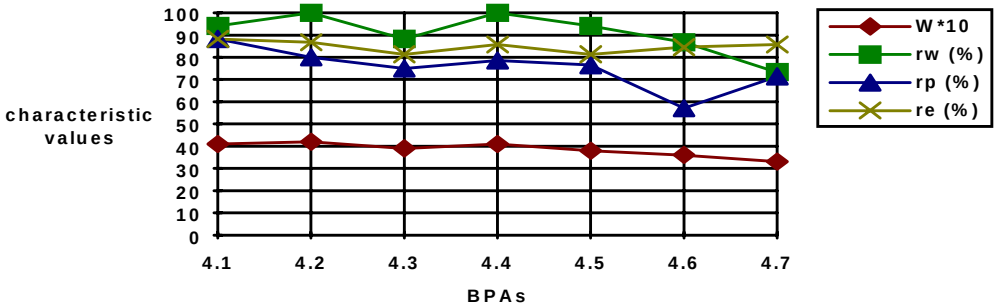


Fig.6 Characteristic curves of the project quality plan process

5. Project management

This section provides statistical data of the survey for 55 BPAs in the project management processes. Based on the benchmark, characteristic curves of the six processes on process management, process tracking, configuration management, change control, process review and intergroup coordinatioin are derived.

5.1 Benchmarked findings on project management processes

This subsection reports the survey findings on the project management processes. Benchmarked results on the mean weighted importance (W), the ratios of significance (r_w), practice (r_p) and effectiveness (r_e), and the characteristic value (φ) for the 55 BPAs in the six processes are listed in Table 3.

Table 3. Benchmark of project management processes

Ref. No.	No.	BPAs	W [0 .. 5]	r_w (%)	r_p (%)	r_e (%)	φ (%)
	1	Process management					
320	1.1	Plan quantitative process management	3.6	94.7	37.5	76.9	27.3
321	1.2	Conduct quantitative process management	3.5	89.5	29.4	64.3	16.9
322	1.3	Collect data for quantitative analysis	3.5	94.7	43.8	71.4	29.6
323	1.4	Control defined process quantitatively	3.4	94.1	37.5	64.3	22.7
324	1.5	Document quantitative analysis results	3.3	78.9	41.2	80.0	26.0
325	1.6	Benchmark organisation's baseline of process capability	2.8	57.9	37.5	71.4	15.5
326	1.7	Manage project by defined process	3.8	94.7	66.7	88.2	55.7
327	1.8	Adopt project/process management tools	3.2	78.9	47.1	70.6	26.2
	2	Process tracking					
328	2.1	Track project progress	4.3	100	100	100	100
329	2.2	Track development schedule	4.2	100	94.1	100	94.1
330	2.3	Track process quality	3.7	100	72.2	66.7	48.1
331	2.4	Track software size	2.9	63.2	68.8	86.7	37.6
332	2.5	Track project cost	3.8	94.4	80.0	93.3	70.5
333	2.6	Track critical resources & performance	3.3	80.0	70.6	94.1	53.1
334	2.7	Track project risks	3.2	84.2	52.9	68.8	30.7
335	2.8	Track process productivity	2.9	68.4	37.5	61.5	15.8
336	2.9	Track system memory utilisation	2.4	44.4	31.3	53.3	7.4
337	2.10	Track system throughput	2.5	55.6	46.7	66.7	17.3
338	2.11	Track system I/O channel capabilities	2.4	58.8	37.5	60.0	13.2
339	2.12	Track system networking	2.5	58.8	33.3	66.7	13.1
340	2.13	Adopt process tracking tools	2.6	55.6	25.0	50.0	6.9
341	2.14	Document project tracking data	3.1	76.5	60.0	73.3	33.6
342	2.15	Identify and handle process deviation	3.7	95.2	78.9	83.3	62.7
	3	Configuration management					
343	3.1	Establish configuration management library	3.8	84.2	77.8	94.4	61.9
344	3.2	Adopt configuration management tools	3.8	93.3	53.3	84.6	42.1
345	3.3	Identify product's configuration	4.2	100	82.4	88.2	72.7
346	3.4	Maintain configuration item descriptions	3.9	93.3	71.4	78.6	52.4
347	3.5	Control change requests	4.4	100	88.2	100	88.2

348	3.6	Release control	4.3	100	81.3	87.5	71.1
349	3.7	Maintain configuration item history	3.9	94.1	68.8	80.0	51.8
350	3.8	Report configuration status	3.6	813	73.3	86.7	51.6
	4	Change control					
351	4.1	Establish change requests/approval system	4.0	100	76.9	100	76.9
352	4.2	Control requirement change	4.1	100	71.4	85.7	61.2
353	4.3	Control design change	3.9	100	71.4	92.9	66.3
354	4.4	Control code change	3.8	93.3	78.6	92.9	68.1
355	4.5	Control test data change	3.3	73.3	57.1	84.6	35.5
356	4.6	Control environment change	3.0	78.6	53.8	81.8	34.6
357	4.7	Control schedule change	3.6	84.6	66.7	100	56.4
358	4.8	Control configuration change	3.8	82.4	73.3	86.7	52.3
359	4.9	Adopt change control tools	2.9	60.0	35.7	76.9	16.5
	5	Process review					
360	5.1	Reviews processes at milestones	3.8	93.8	80.0	84.6	63.5
361	5.2	Document project review data	3.6	80.0	64.3	69.2	35.6
362	5.3	Revise project process	3.7	80.0	57.1	85.7	39.2
363	5.4	Conduct statistical analysis of process	3.1	68.8	42.9	61.5	18.1
364	5.5	Gather process data	3.2	71.4	61.5	66.7	29.3
365	5.6	Compare actual/forecasted errors	3.5	86.7	57.1	76.9	38.1
366	5.7	Compare actual/forecasted schedule	4.1	100	57.1	92.9	53.1
367	5.8	Compare actual/forecasted resources	4.0	100	46.2	76.9	35.5
	6	Intergroup coordination					
368	6.1	Define interface between project groups	3.7	73.3	66.7	80.0	39.1
369	6.2	Plan intergroup activities	3.6	87.5	66.7	92.9	54.2
370	6.3	Identify intergroup critical dependencies	3.8	81.3	53.3	86.7	37.6
371	6.4	Handle intergroup issues	3.8	88.2	68.8	81.3	49.3
372	6.5	Technical/management representatives coordination	3.6	94.1	75.0	93.3	65.9
373	6.6	Review last process's output	3.1	75.0	40.0	78.6	23.6
374	6.7	Conduct intergroup representatives review	3.5	86.7	64.3	85.7	47.8

5.2 Process management

The characteristic curves of the process management process are derived in Fig.7. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 1.1 and 1.7. Almost all the BPAs in this process need to be improved because of lacking of practices.

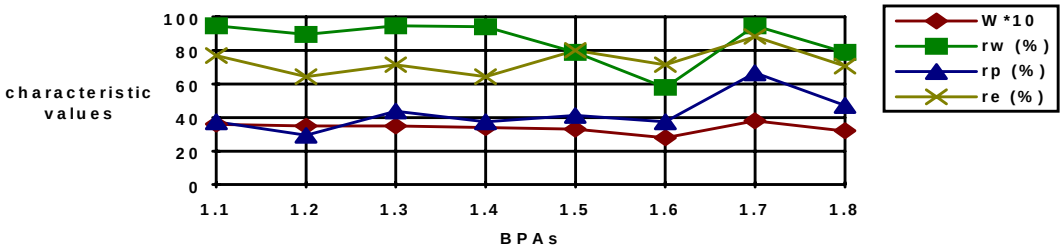


Fig.7 Characteristic curves of the process-management process

5.3 Process tracking

The characteristic curves of process tracking are derived in Fig.8. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 2.1, 2.2, 2.5 and 2.15. For process improvement, the priority can be put on the BPAs 2.3, 2.7, 2.8 and 2.13 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

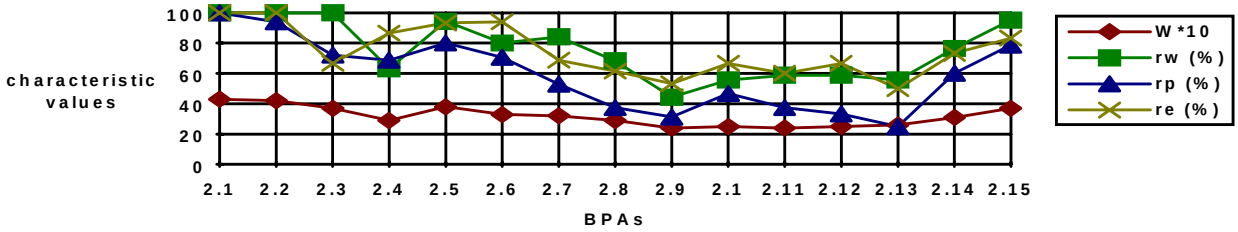


Fig.8 Characteristic curves of the process-tracking process

5.4 Configuration management process

The characteristic curves of the configuration management process are derived in Fig.9. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 3.3, 3.5 and 3.6. For process improvement, the priority can be put on the BPAs 3.2 and 3.7 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

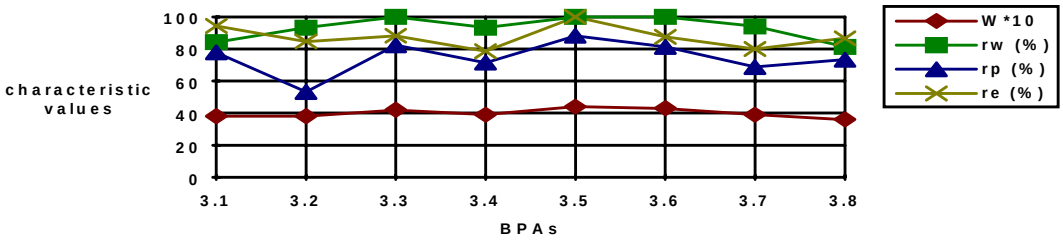


Fig.9 Characteristic curves of the configuration management process

5.5 Change control process

The characteristic curves of the change control process are illustrated in Fig.10. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 4.1, 4.3 and 4.4. For process improvement, the priority can be put on the BPAs 4.2, 4.3, 4.6 and 4.9 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

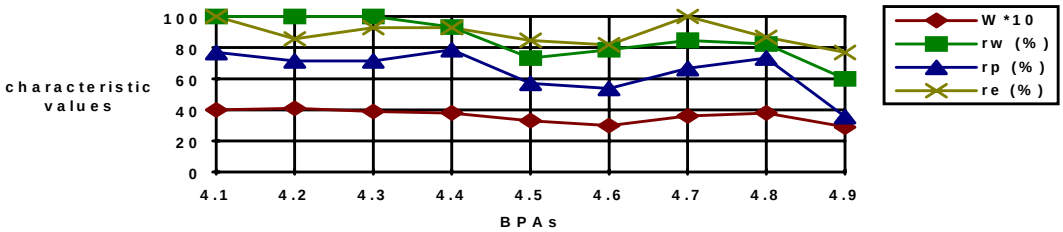


Fig.10 Characteristic curves of the change control process

5.6 Process review

The characteristic curves of process review are derived in Fig.11. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 5.1 and 5.7. For process improvement, the priority can be put on the BPAs 5.3, 5.6, 5.7 and 5.8 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

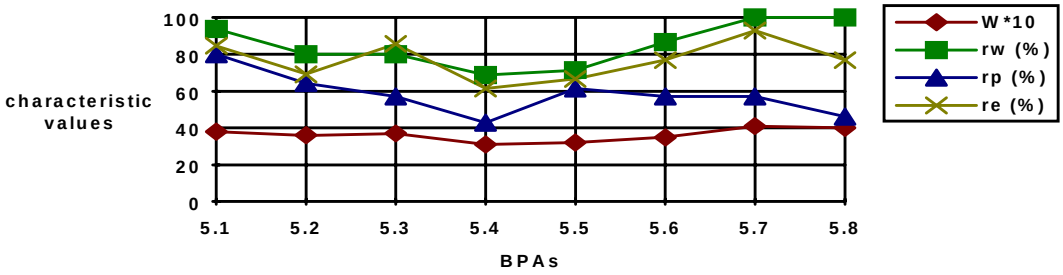


Fig.11 Characteristic curves of the process-review process

5.7 Intergroup coordination process

The characteristic curves of the intergroup coordination process are derived in Fig.12. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 6.5 and 6.4. For process improvement, the priority can be put on the BPAs 6.3 and 6.6 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

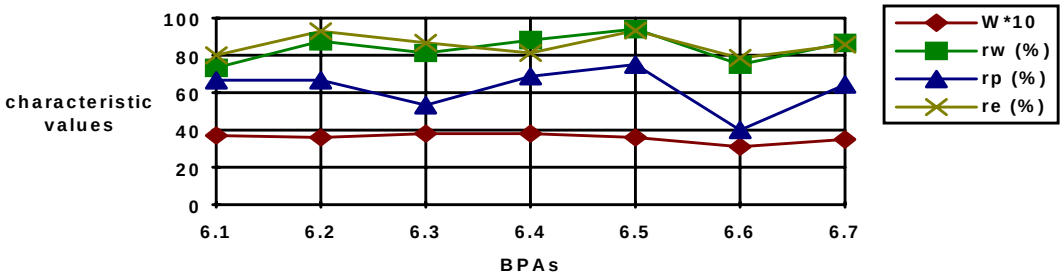


Fig.12 Characteristic curves of the intergroup coordination process

6. Contract and requirement management

This section provides statistical data of the survey for 42 BPAs in the four contract and requirement management processes. Based on the benchmark, characteristic curves of the processes on requirement management, contract management, subcontractor management and purchasing management are derived.

6.1 Benchmarked findings on contract and requirement management processes

This subsection reports the survey findings on the contract and requirement management processes. Benchmarked results on the mean weighted importance (W), the ratios of significance (r_w), practice (r_p) and effectiveness (r_e), and the characteristic value (φ) for the 42 BPAs in the four processes are listed in Table 4.

Table 4. Benchmark of contract and requirement management processes

Ref. No.	No.	BPAs	W [0 .. 5]	r_w (%)	r_p (%)	r_e (%)	φ (%)
	1	Requirement management					
375	1.1	Specify system requirements	4.7	100	100	94.1	94.1
376	1.2	Design system based on requirements	4.4	100	100	86.7	86.7
377	1.3	Allocate requirements	3.8	86.7	85.7	85.7	63.7
378	1.4	Determine operating environment impact	3.6	94.1	68.8	86.7	56.1
379	1.5	Determine software requirements	4.5	100	100	93.8	93.8
380	1.6	Analysis software requirements	4.3	100	93.8	87.5	82.1
381	1.7	Evaluate requirements with customer	4.2	100	100	100	100
382	1.8	Update requirements for next iteration	3.8	81.3	80.0	80.0	52.0
383	1.9	Agree on requirements	4.3	93.8	93.8	93.3	82.0
384	1.10	Establish requirements standard	3.7	76.5	68.8	73.3	38.6
385	1.11	Manage requirements changes	4.1	94.1	75.0	87.5	61.8
386	1.12	Maintain requirements traceability	3.8	93.8	62.5	80.0	46.9
	2	Contract management					
387	2.1	Define contractual procedures	3.9	93.8	87.5	100	82.0
388	2.2	Prepare contract proposal	3.6	87.5	93.3	100	81.7
389	2.3	Review contract	3.9	100	70.6	85.7	60.5
390	2.4	Ensure agreement of terminology	4.0	100	56.3	93.3	52.5
391	2.5	Determine interfaces to independent agents	3.3	80.0	46.7	85.7	32.0
392	2.6	Assess contractor's capability	3.7	87.5	50.0	92.3	40.4
393	2.7	Document contractor's capability	3.2	75.0	50.0	91.7	34.4
	3	Subcontractor management					
394	3.1	Specify subcontracted development	4.0	100	85.7	92.3	79.1
395	3.2	Assess capability of subcontractors	3.9	93.8	80.0	100	75.0
396	3.3	Record acceptable subcontractors	3.5	80.0	57.1	76.9	35.2
397	3.4	Define scope of contracted work	4.0	93.8	80.0	86.7	65.0
398	3.5	Define interface of contracted work	3.9	100	92.3	100	92.3
399	3.6	Select qualified subcontractor	3.9	100	73.3	93.3	68.4
400	3.7	Approve subcontractor's plan	3.5	80.0	50.0	84.6	33.8
401	3.8	Maintain interchanges with subcontractors	3.6	93.3	78.6	85.7	62.9
402	3.9	Track subcontractor's development activities	3.2	78.6	41.7	72.7	23.8
403	3.10	Monitor subcontractor's SQA activities	3.4	86.7	50.0	92.3	40.0
404	3.11	Review subcontractor's work	3.5	93.3	78.6	92.3	67.7
405	3.12	Assess compliance of contracted product	4.2	100	85.7	100	85.7
406	3.13	Determine interfaces to subcontractors	3.5	92.3	76.9	83.3	59.2
407	3.14	Document subcontractor's records	2.9	66.7	50.0	66.7	22.2
	4	Purchasing management					
408	4.1	Identify need of purchasing	3.7	93.3	92.9	92.9	80.5
409	4.2	Define purchasing requirements	3.7	87.5	93.3	93.3	76.2

410	4.3	Prepare acquisition strategy	3.0	61.5	53.8	72.7	24.1
411	4.4	Prepare purchasing document	2.9	64.3	71.4	84.6	38.9
412	4.5	Prepare request for proposal	3.1	76.9	69.2	90.0	47.9
413	4.6	Review purchasing document	3.3	73.3	66.7	84.6	41.4
414	4.7	Select software product supplier	3.8	92.9	85.7	92.3	73.5
415	4.8	Verify purchased product	4.1	100	71.4	100	71.4
416	4.9	Manage purchased tools configuration	3.0	71.4	50.0	75.0	26.8

6.2 Requirement management

The characteristic curves of the requirement management process are derived in Fig.13. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 1.1, 1.2, 1.5-1.7 and 1.9. For process improvement, the priority can be put on the BPAs 1.4 and 1.12 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

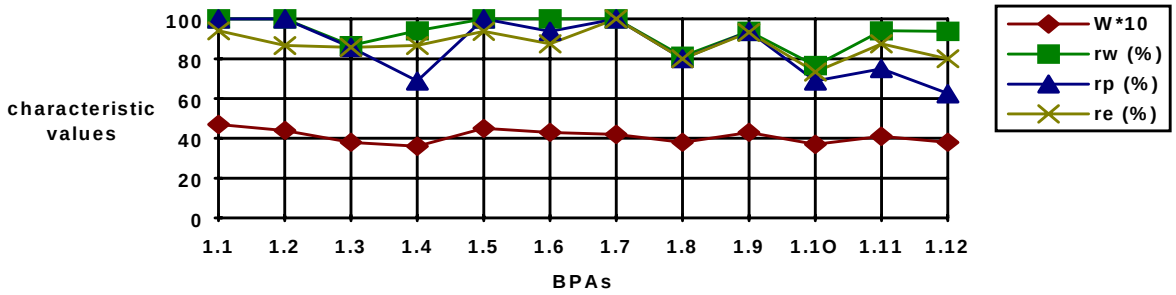


Fig.13 Characteristic curves of the requirement management process

6.3 Contract management

The characteristic curves of the contract management process are derived in Fig.14. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 2.1 and 2.4. For process improvement, the priority can be put on the BPAs 2.3- 2.6 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

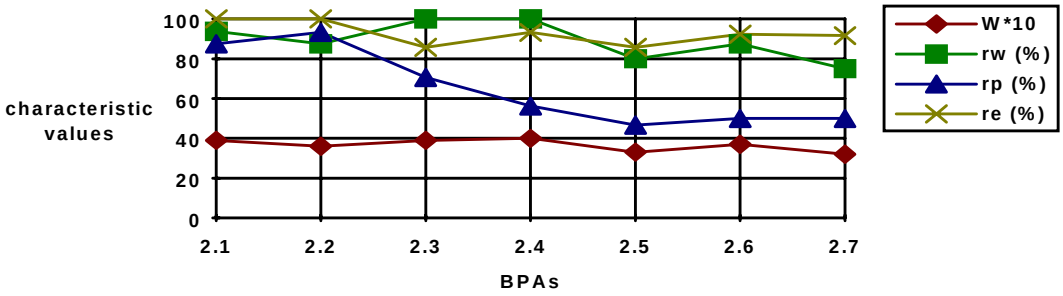


Fig.14 Characteristic curves of the contract management process

6.4 Subcontractor management

The characteristic curves of the subcontractor management process are derived in Fig.15. For process establishment, the implementation priority can be put on the BPAs with higher ratio of

significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 3.1, 3.5, 3.6 and 3.12. For process improvement, the priority can be put on the BPAs 3.3, 3.7 and 3.10 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

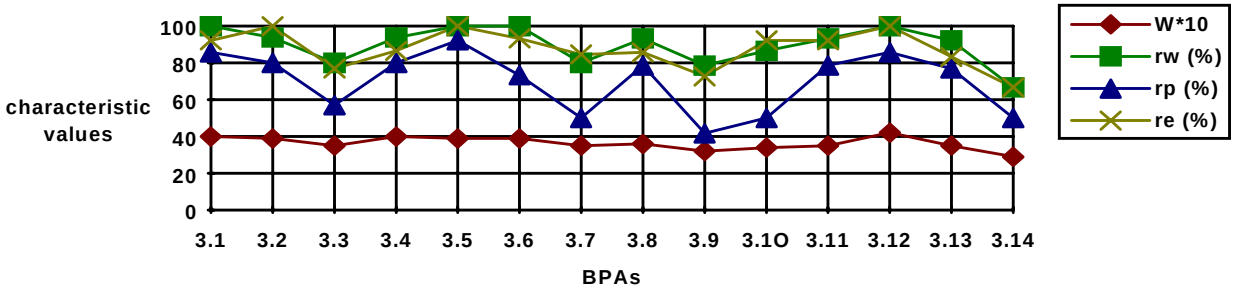


Fig.15 Characteristic curves of the subcontractor management process

6.5 Purchasing management

The characteristic curves of the purchasing management process are derived in Fig.16. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 4.1, 4.2, 4.7 and 4.8. For process improvement, the priority can be put on the BPAs 4.8 and 4.9 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

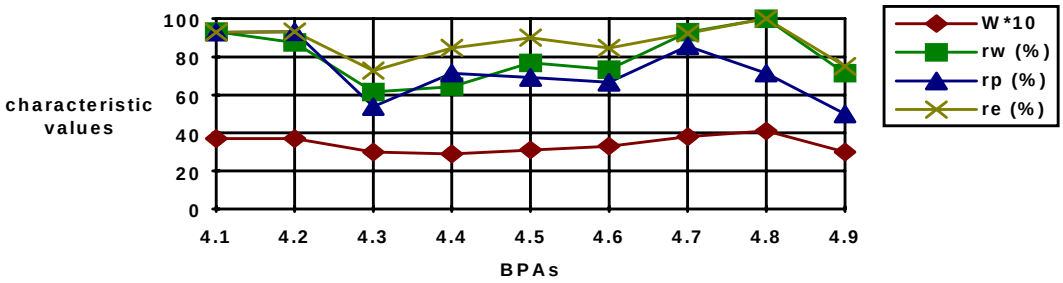


Fig.16 Characteristic curves of the purchasing management process

7. Document management

This section provides statistical data of the survey for 17 BPAs in the two document management processes. Based on the benchmark, characteristic curves of the documentation and process database/library processes are derived.

7.1 Benchmarked findings on document management processes

This subsection reports the survey findings on the document management processes. Benchmarked results on the mean weighted importance (W), the ratios of significance (r_w), practice (r_p) and effectiveness (r_e), and the characteristic value (φ) for the 17 BPAs in the two processes are listed in Table 5.

Table 5. Benchmark of document management processes

Ref. No.	No.	BPAs	W [0 .. 5]	r_w (%)	r_p (%)	r_e (%)	φ (%)
	1	Documentation					
417	1.1	Master list of project documents	3.8	88.2	81.3	86.7	62.1
418	1.2	Determine documentation requirements	3.5	88.2	68.8	81.3	49.3
419	1.3	Develop document	4.2	100	86.7	93.3	80.9
420	1.4	Check document	3.9	100	78.6	85.7	67.3
421	1.5	Control document issue	3.8	89.5	66.7	82.4	49.1
422	1.6	Maintain document	3.8	94.4	87.5	82.4	68.1
423	1.7	Documentation according to defined process	3.5	76.5	81.3	82.4	51.2
424	1.8	Establish documentation standards	3.8	88.2	81.3	86.7	62.1
425	1.9	Safety document storage	3.0	62.5	60.0	66.7	25.0
426	1.10	Identify current version of documents	4.0	94.1	86.7	87.5	71.4
427	1.11	Adopt interactive documentation tools	2.9	64.7	53.3	62.5	21.6
	2	Process database/library					
428	2.1	Establish organisation's process library	3.1	68.8	40.0	78.6	21.6
429	2.2	Establish organisation's process database	3.1	73.3	28.6	72.7	15.2
430	3.3	Establish software reuse library	3.3	60.0	33.3	80.0	16.0
431	4.4	Establish organisation's metrics database	3.6	70.6	43.8	80.0	24.7
432	5.5	Establish operation manual library	3.1	73.3	61.5	92.3	41.7
433	6.6	Establish practice benchmark database	2.3	50.0	0	58.3	0

7.2 Documentation process

The characteristic curves of the documentation process are derived in Fig.17. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 1.3, 1.4, 1.6 and 1.10. For process improvement, the priority can be put on the BPAs 1.2 and 1.5 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

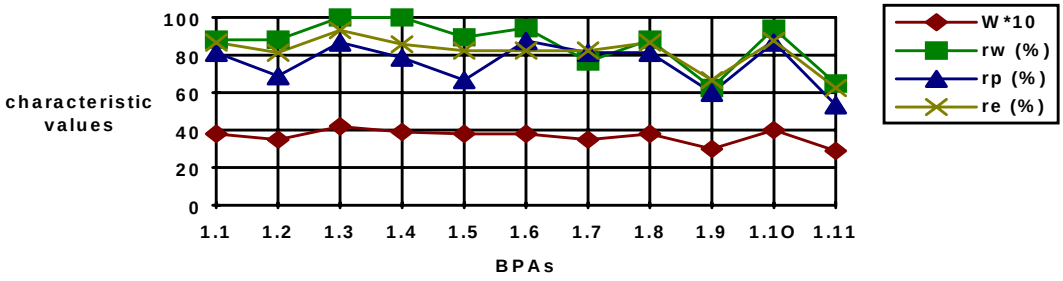


Fig.17 Characteristic curves of the documentation process

7.3 Process database/library

The characteristic curves of process database/library are derived in Fig.18. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 2.2 and 2.5. For process

improvement, the priority can be put on the BPAs 2.2, 2.4 and 2.6 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

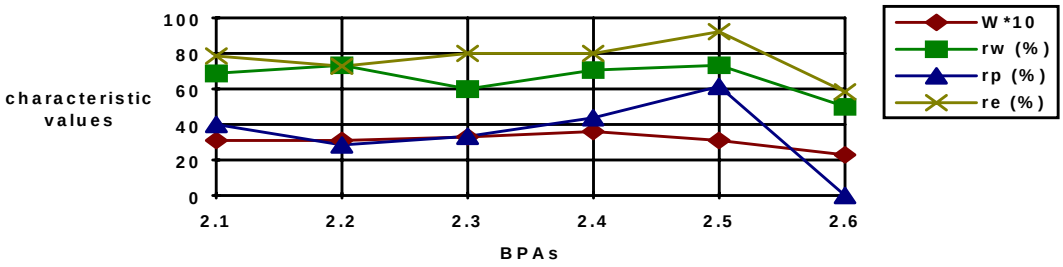


Fig.18 Characteristic curves of process database/library

8. Human resource management

This section provides statistical data of the survey for 11 BPAs in the human resource management processes. Based on the benchmark, characteristic curves of the staff selection/ allocation and training processes are derived.

8.1 Benchmarked findings on human resource management processes

This subsection reports the survey findings on the human resource management processes. Benchmarked results on the mean weighted importance (W), the ratios of significance (r_w), practice (r_p) and effectiveness (r_e), and the characteristic value (φ) for the 11 BPAs in the two processes are listed in Table 6.

Table 6. Benchmark of human resource management processes

Ref. No.	No.	BPAs	W [0 .. 5]	r_w (%)	r_p (%)	r_e (%)	φ (%)
	1	Staff selection and allocation					
434	1.1	Define qualifications for positions	4.1	92.3	83.3	91.7	70.5
435	1.2	Define experience for positions	3.9	92.9	78.6	85.7	62.5
436	1.3	Assign personnel selection group	3.3	92.9	76.9	92.9	66.3
437	1.4	Select staff by qualification /experience	4.1	100	85.7	92.9	79.6
	2	Training					
438	2.1	Plan training	3.9	100	85.7	85.7	73.5
439	2.2	Identify training needs	4.1	100	93.8	93.8	87.9
440	2.3	Develop training courses	3.5	85.7	71.4	85.7	52.5
441	2.4	Approval training courses	3.2	78.6	38.5	90.0	27.2
442	2.5	Conduct technical training	4.1	100	76.5	87.5	66.9
443	2.6	Conduct management training	3.7	92.9	50.0	78.6	36.5
444	2.7	Document training records	3.6	82.4	81.3	93.3	62.5

8.2 Staff selection and allocation process

The characteristic curves of the staff selection and allocation process are derived in Fig.19. For process establishment, the implementation priority need to be put on all the BPAs in this process.

For process improvement, the priority can be put on the BPA 1.3 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

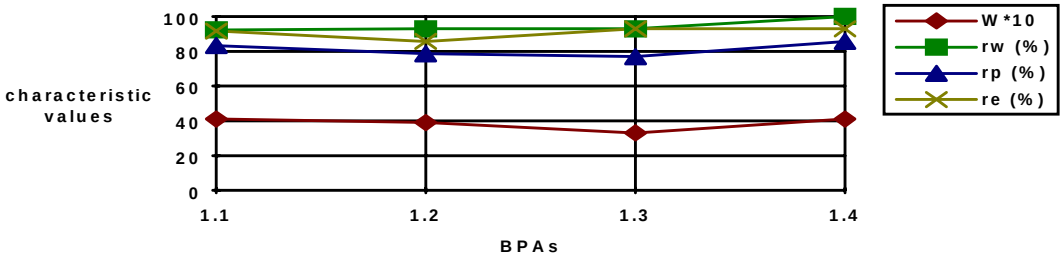


Fig.19 Characteristic curves of the staff selection and allocation process

8.3 Training process

The characteristic curves of the training process are derived in Fig.20. For process establishment, the implementation priority can be put on the BPAs with higher ratio of significance (r_w) and ratio of effectiveness (r_e), such as the BPAs 2.1, 2.2, and 2.5. For process improvement, the priority can be put on the BPAs 2.4 and 2.6 which have the largest gaps between the current practices (r_p) and the ratio of significance (r_w).

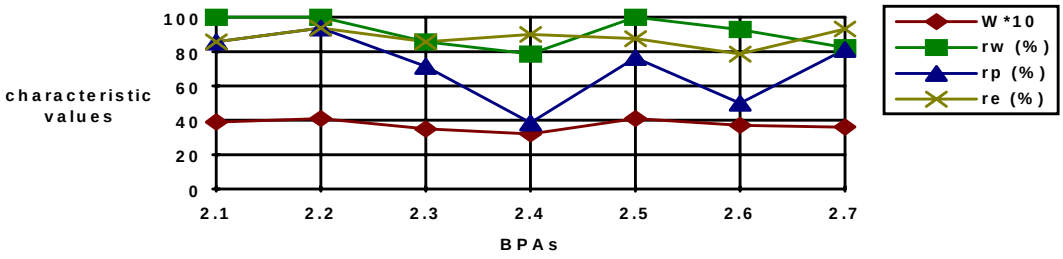


Fig.20 Characteristic curves of the training process

9. Other findings

This section provides additional information found in the survey for the general distributions of profession, geography, applied standard/model and business scale.

9.1 Professional distribution

The professional distribution of the survey is shown in Fig.21. In descending order they are SQA manager/engineer, senior manager, software engineer, academic, consultant, project manager and system analyst. The first three are composed more than half of the sample population.

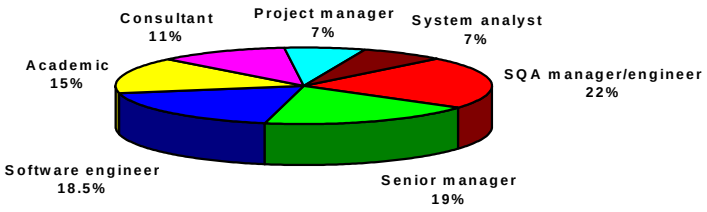


Fig.21 Professional distribution

9.2 Geographical distribution

The geographical distribution of the survey is shown in Fig.22. North America, Europe and Asia-Pacific region are about evenly distributed in the sample space.

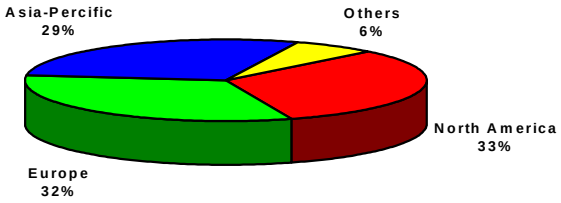


Fig.22 Geographical distribution

9.3 Applied standard/model distribution

The applied standard/model distribution of the survey is shown in Fig.23. It is interesting to find that ISO 9000 series are still the most popular registered standards followed by the CMM and SPICE. It is noteworthy that the regional, internal and industry sector process models, such as the Trillum etc, also share a significant part in the survey.

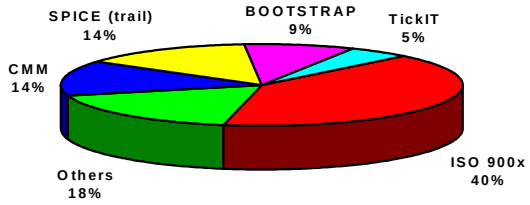


Fig.23 Applied standard/model distribution

9.4 Business scale distribution

The business scale distribution is shown in Fig.24. Almost 95% of the software organizations are small or medium sized. This figure indicates that the process models and standards have to cover the needs of all scaled software organizations. Therefore tailorability is important in modelling of the software engineering processes in standardization.

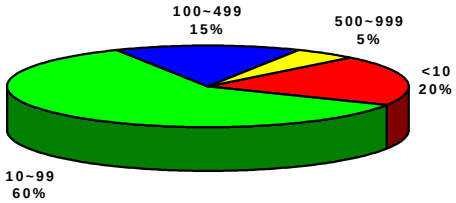


Fig.24 Business scale distribution

10. Conclusions

This paper reports the survey findings on the software management process activities. It is a subset of the benchmark of the worldwide survey on BPAs towards software process excellence. In this paper, the design of the survey has been introduced. The data processing criteria and methods of the survey are formally described. A detailed benchmark of the 170 BPAs in the 18 management processes and five categories are obtained for the attributes of the mean weighted importance and the ratios of significance, practice and effectiveness.

New BPAs found in the survey, which are not modelled in the existing models and standards, can be categorised into the following groups: a) Evaluate the life cycle, prototype, OOP, combined and CASE models; b) Adopt supporting tools of software design, CASE, requirement acquisition, testing, SQA, requirement review, design review, testing analysis, configuration management, documentation processing, and specification verification; and c) Establish reuse library, operation manual library, and practice benchmark database, etc. The survey indicates that the above BPAs are useful, important, effectiveness and practical.

Table 7. General statistics of the survey findings

	Mean weighted importance (W)						Ratio of significance (r_w)				Ratio of Practice (r_p)				Ratio of effectiveness (r_e)			
	0	1	2	3	4	5	E	V	F	N	E	V	F	N	E	V	F	N
No. of BPAs	0	0	16	114	40	0	80	70	19	1	26	61	47	36	57	89	24	0
Ratio (%)	0	0	39.4	67.1	23.5	0	47.0	41.2	11.2	0.6	15.3	35.9	27.6	21.2	33.5	52.4	14.1	0

Note: E - Extremely (≥90%), V - very (70-89%), F - fairly (50-69%), and N - not (<50%)

It is interesting to find, as shown in Table 7 and Fig.25, that:

- On the mean weighted importance of the BPAs in software management processes, 90.6% of the BPAs are heavily weighted with 67.1% at weight scale 3.0-3.99 and 23.5% at weight scale 4.0-4.99. There are about one third BPAs perceived to be not very important;

- On the ratio of significance of the BPAs, 47.0% of the BPAs are weighted extremely significant, 41.2% are very significant, 11.2% are fairly significant, and only 0.6% are not significant;

- On the ratio of practice of the BPAs, 15.3% BPAs have got extremely high application rate, 35.9% BPAs have very high application rate, and 27.6% have fairly high application rate. But

it is noteworthy there are 21.2% BPAs which were lack of practice; and

- On the ratio of effectiveness of the BPAs, 33.5% of the BPAs are weighted extremely effective, 52.4% are very effective, and 14.1% are fairly effective. No BPAs in the set are found not effective.

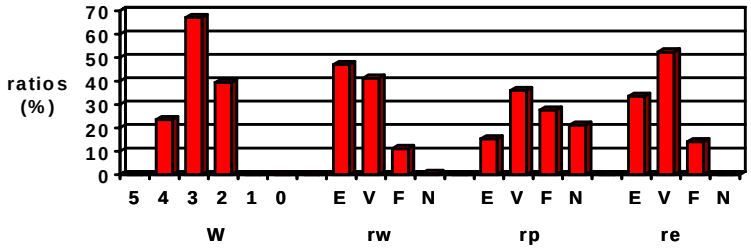


Fig.25 Overview of the survey findings

The survey results shown in Tables 2~7 provide a set of valuable statistical data. The benchmark is useful for modelling and feature-identifying the fundamental software process activities in software management practices; for evaluating a software organization's current practice gaps to the benchmarks; and for identifying process improvement opportunities for an organization's software management processes.

Acknowledgements

The authors would like to acknowledge the support of the IVF Centre for software Engineering, European Software Institute and BCS QSig.

References

[1] Wang, Y., Court, I., Ross, M., Staples, G. and King, G. [1996], Towards a Software Process Reference Model (SPRM*), Proceedings of International Conference on Software Process Improvement (SPI'96),* Brighton UK, pp.145-166.

[2] Wang, Y., Court, I., Ross, M., Staples, G., King, G. and Dorling, A. [1997], Quantitative Analysis of Compatibility and Correlation of the Current SPA/SPI Models, Proceedings of the 3rd IEEE International Symposium on Software Engineering Standards (ISESS'97), IEEE Computer Society Press, USA, pp.36-56.

[3] Wang, Y., Court, I., Ross, M., Staples, G., King, G. and Dorling, A. [1997], Quantitative Evaluation of the SPICE, CMM, ISO 9000 and BOOTSTRAP, Proceedings of the 3rd IEEE International Symposium on Software Engineering Standards (ISESS'97), IEEE Computer Society Press, USA, pp.57-68.

[4] Wang, Y., Court, I., Ross, M., Staples, G., King, G. and A. Dorling [1997], Design of the Questionnaires for the Survey of Base Process Activities Towards Software Process Excellence, *Technical Report SI-RCSE-WANG96-DATA03*, pp.1-36.

[5] Wang, Y., Court, I., Ross, M., Staples, G., King, G. and A. Dorling [1997], Analysis Report of the Survey of Base Process Activities Towards Software Process Excellence, *Technical Report SI-RCSE-WANG97-PROC260*, pp.1-27.
[6] Wang, Y., Court, I., Ross, M., Staples, G., King, G. and A. Dorling [1997], A Survey of Base Process Activities Towards Software Process Excellence (I) - Organisation Processes, *Technical Report SI-RCSE-WANG97-PROC261*, pp.1-38.
[7] Wang, Y., Court, I., Ross, M., Staples, G., King, G. and A. Dorling [1997], A Survey of Base Process Activities Towards Software Process Excellence (II) - Development Processes, *Technical Report SI-RCSE-WANG97-PROC262*, pp.1-42.
[8] Wang, Y., Court, I., Ross, M., Staples, G., King, G. and A. Dorling [1997], A Survey of Base Process Activities Towards Software Process Excellence (III) - Management Processes, *Technical Report SI-RCSE-WANG97-PROC263*, pp.1-56.
[9] ISO/IEC JTC1/SC7/WG10 [1996], Software process assessment - Part 2: A reference model for processes and process capability (V.2.0), pp. 1 - 38.
[10] ISO/IEC JTC1/SC7/WG10 [1996], Software process assessment - Part 5: An assessment model and indicator guidance (1st PDTR), pp.1-130.
[11] ISO/IEC JTC1/SC7/WG10 [1996], Software process assessment - Part 5: An assessment model and indicator guidance (V.2.0), pp.1-138.
[12] Humphrey, W.S. and W.L. Sweet [1987], A Method for Assessing the Software Engineering Capability of Contractors, Technical Report CMU/SEI-87-TR-23, Software Engineering Institute, Pittsburgh, Pennsylvania, USA.
[13] Paulk, M.C., Curtis, B., Chrissis, M.B. and Weber, C.V. [1993], Capability Maturity Model for Software, Version 1.1, Software Engineering Institute, CMU/SEI-93-TR-24.
[14] Paulk, M.C., Weber, C.V., Garcia, S., Chrissis, M.B. and Bush, M. [1993], Key Practices of the Capacity Maturity Model, Version 1.1, Software Engineering Institute, CMU/SEI-93-TR-25.
[15] International Organisation for Standardisation [1994], Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation, and Servicing, ISO 9001, Revised Edition.
[16] International Organisation for Standardisation [1991], Quality Management and Quality System Elements (Part 3) - Guidelines for Development, Supply and Maintenance of Software, ISO 9000-3.
[17] Jenner, M.J. [1995], Software Quality Management and ISO 9001, John Wiley & Sons, Inc., pp. 47-160, 223-234.
[18] Kuvaja, P., Simila, J., Kizanik, L., Bicego, A., Koch, G. and Saukkonen, S. [1994], Software Process Assessment and Improvement: The BOOTSTRAP Approach, Blackwell Business Publishers.
[19] Haase, V., Messmarz, R., Koch, G., Kugler, H.J. and Decrinis, P., [1994], Bootstrap: Fine-Tuning Process assessment, IEEE Software, July issue, pp.25-35.

AUTOMATED TEST GENERATION, EXECUTION, AND REPORTING

Sadik Esmelioglu, Ph.D.
Lucent Technologies Inc.
101 Crawfords Corner Road Rm:2K-236
Holmdel, NJ 07733
Phone: (908) 949-3636
Fax: (908) 949-9650
Email: sadik@lucent.com

Larry Apfelbaum
Teradyne Software & Systems Test
44 Simon Street
Nashua, NH 03060
Phone: (603) 791-3555
Fax: (603) 791-3075
Email: larry@sst.teradyne.com

ABSTRACT

This paper describes how the regression test cycle has been reduced by using TestMaster, a Test Generation System (TGS), and BUSTER, a Test Management System (TMS) in a large software application.

Test automation reduces the time and cost of testing a product, however, tremendous amount of effort is needed to maintain automated test scripts. As specifications change the test suites also need to be changed, requiring testers to add new test cases as well as delete and modify existing test cases. Automating the test generation process reduces this maintenance cost dramatically. TestMaster is used to generate test cases through the use of a functional model.

BUSTER provides a harness to store, retrieve and execute automated test cases enabling testers to select and execute suites of test cases easily. In addition, BUSTER stores the test results in a database. Scripts are written to produce test reports providing testers and project managers the capability to track the progress of the testing activity.

KEYWORDS

Test Case Generation, Test Automation, Web Reporting

BIOGRAPHY

Dr. Sadik Esmelioglu is a Technical Manager at Lucent Technologies responsible for System Test and Customer Support. He has been involved in testing and supporting software applications for a range of domestic and international telecommunications systems. Prior to joining Lucent (formerly part of AT&T), Dr. Esmelioglu has taught at Electrical and Computer Engineering department of Auburn University, AL. He received his B.S. in Electrical Engineering at Bogazici University in Istanbul, Turkey, and M.S. and Ph.D. degrees in Electrical and Computer Engineering at the University of South Carolina in Columbia, SC.

Larry Apfelbaum is the manager of Teradyne's Software and Systems Test division. Prior to joining this group, he managed product teams developing a computer aided engineering tool suite focused on hardware design and test. He has been with Teradyne since 1973 and has been involved in the development and support of automatic test systems and automated test generation solutions utilized in modern manufacturing environments. Mr. Apfelbaum holds a Bachelor's Degree in Electrical Engineering (1973) and a Master's Degree in Computer Science (1973) from Rensselaer Polytechnic Institute.

1. Introduction

1.1 Problem Description

Application Under Test (AUT) is an Order Management System used by telecommunication companies to track the service orders. Frequent releases and changing requirements require extensive regression testing. When schedules demand quick turnaround, regression testing suffers, resulting in broken functionality which used to work released to the customers undetected. In order to improve the quality and reduce the testing cycle time, regression test cases can be automated. However, in addition to the initial investment of automating test cases, constantly maintaining them may turn out to be more costly. The effort to keep automated regression test cases up to date with changing functionality requires extra cycles which the project does not have. Therefore, it is not enough to automate test cases, but the generation of automated test cases itself needs to be automated so that the maintenance is minimal.

The test group for this project made use of an internally developed tool[1], which used a model based approach to generate test cases. However, the tool was developed with one of the subsystems in mind and to make it applicable to the other subsystems of the application required significant effort. Since the developer of the tool left the company, it would have required significant effort to extend the coverage of the tool and maintain it. As the software evolved that the upkeep on the tool fell behind, much of the automated process reverted back to manual tasks. This process consumed most of the tester's time. Also, the testers kept track of the status of their test cases themselves. In order to come up with a test status report, the project manager then would go to each tester, gather the results and tabulate them. This process posed an overhead for the testers as well as the project manager and the results were often inconsistent and not accurate from week to week.

1.2 Goals and Obstacles

The major goal of the project was to reduce the cost/cycle-time of regression testing, so that all of the regression test cases can be executed within the time allotted and with less resources (or work regular hours). After analyzing where most of the time was spent, the following objectives were set to reach the goal:

1. Test cases should be automatically generated from the feature description or model so that when the functionality changes, updating the regression test suite is accomplished easily and in a short time
2. The generated test cases should be in a form that can be run in an automated fashion
3. The results of the test execution should be interpreted and recorded automatically and readily available to testers and the project manager

In trying to find a solution it was important that enough resources and budget were allocated for this effort. After sharing the plan with the stake-holders, the required backing was received for a pilot project.

1.3 Plan and Countermeasures

Before full implementation, it was decided to pilot the project on a subsystem of the application which was small enough for a one-month project, but still represented the complexity of the product. Work Order (WO) was selected as the subsystem for the pilot, TestMaster was chosen as the Test Generation System (TGS) and BUSTER was chosen as the Test Management System (TMS). Other subsystems would be added to the automation one by one if the pilot project was successful.

2. Application Under Test (AUT)

The following subsections describe the Work Order (WO) subsystem which is piloted, the test environment and the test methodology used.

2.1 Work Order (WO) Subsystem

A simplified high level functionality of the WO subsystem is shown in Figure 1. Each state in the diagram is modeled in detail, but the lower level views are not discussed in this paper.

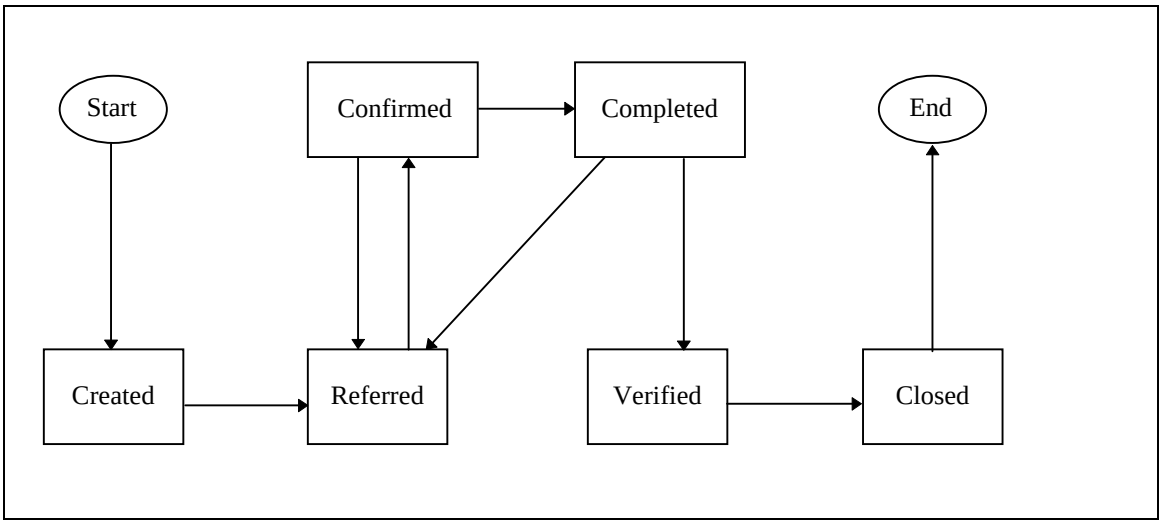


Figure 1: Work Order High Level State Transition Diagram

When a work order is received, an internal ID is assigned, necessary entries in the database are created, and the order is picked up by an operator. The operator then refers the work to the appropriate technician who then confirms or refers it to another person. Once the work is completed it may be still be referred to another technician for additional work or sent for verification. After verification, the order can be closed. Some of the work orders are handled through automated interfaces and do not require any human intervention.

2.2 Test Environment

The test environment for the purpose of the pilot consists of simply a server running Solaris 2.5 where the AUT resides and a test tool server running SunOS 4.1.3 where BUSTER resides. These two machines are located on the same LAN.

2.3 Test Methodology

Since the application User Interface is developed and tested by a different group, only the mainline processing of the work orders from creation to closure are tested. The testing is accomplished by moving a work order from one state to the other using an input file and a utility called *ud* to make the transition as shown below:

ud < transition.input.file > transition.output.file

In order to test a scenario from *start* to *end*, input files for each transition need to be prepared. The input files consist of name-value pairs which describe the work order and the state it needs to be transitioned to. The *ud* script is then applied to each input file in the sequence of the scenario. The output of the utility is examined after each step to see if it was successful. In addition, database entries are retrieved using a *get* input file with the utility and the output file is examined for correctness.

The input files can be used for future testing, but reuse requires two parameters to be modified every time they are used: 1) The internal Work Order ID which is assigned automatically when a new order is created and 2) Commit Time which should be same as or later than the current date.

3. Automated Test Generation

3.1 Developing a finite-state model of the application to be tested

There are several approaches that can be used to develop tests from a model of an application. Central to most of these is the concept of a path. A path is a sequence of events or actions that traverse through the model defining an

actual user scenario of the system. Each element in a path, a transition or state, can have some testing information associated with it. The information defines what test actions are required to move the system from its current state to the next state, verifies that the state is reached or checks that the system has responded properly to previous inputs. Once a path through the model has been defined, a test script can be created for that path. When this script is applied to the actual system, the actual system follows the same sequence (or path) as defined by the model path from which the test script was extracted. This process can then be repeated for another path, which defines another user scenario, and verifies another sequence of actions. Many methods can be used to select paths, each with its own distinct objectives and advantages. The test objective was for two sets of tests, one to verify basic functionality the other to provide a complete product verification suite.

The testing environment consists of three different testing components, each with a unique set of requirements for the test generation process. A single model is used to produce :

- Scripts for functional tests
- Input files required for each transaction
- BUSTER header information and build process for each test

3.2 Modeling the Work Order Application

The Work Order application is specified by a Feature Requirements Specification, a set of operational scenarios and process flows, and database definitions. All of these sources are used to derive the information necessary to build a model. A forms-based system can be described as a series of transactions in a model; States, Transitions and Events are used to represent the behavior of the system. Data structures within the model are used to represent the database record of the work order, and conditional instructions are used to model the transaction semantics (i.e. dependencies) of the application. The model is built from the perspective of a user of the system. All of the actions that a user can invoke are represented graphically via arrows in the model.

Figure 2 depicts the top level TestMaster model that represents the processing of a work order. It shows various states and transitions that a work order goes through.

The modeling effort for the pilot took 10 person-days to complete. A major goal of the modeling effort was to provide a model of the AUT which was easy to use and maintain. This was accomplished by providing some standard “model templates” and applying these templates for each transaction. This resulted in a slightly larger model, but one which proved to be very easy and efficient to extend and maintain. A summary of the model is shown in Table 1.

Number of AUT Transactions Modeled	15
Number of Models Created	216
Total Number of States	500
Total Number of Transitions	900
Person-Days to Create and Constrain the Model	10

Table 1: Model Characteristics

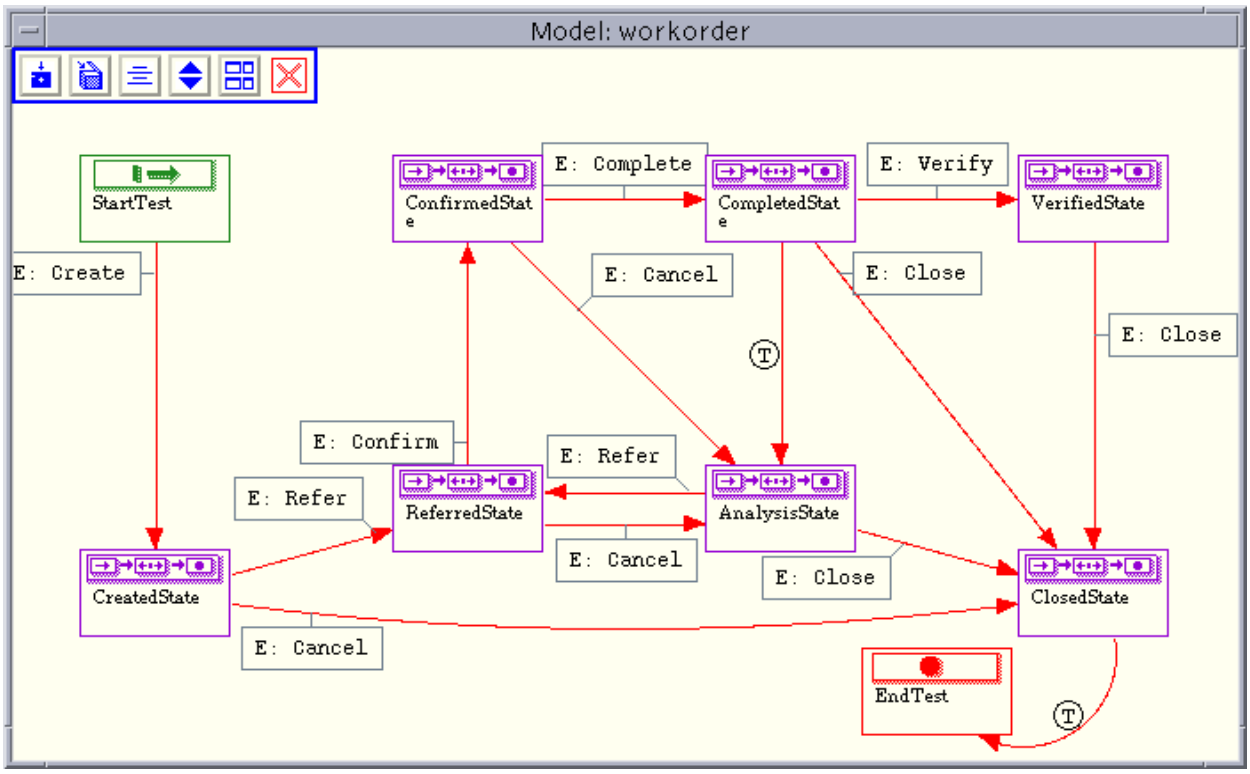


Figure 2: Top Level Model of Work Order Application

3.3 Modeling behavior to generate functional tests

The Functional Requirements Specification and process flow documents define the behavior of the application. The AUT in this case is a forms-based system that builds a record and submits a transaction. The documents contain functional descriptions of the application as well as detailed data descriptions. The specifications are used at the highest level to determine valid user scenarios for processing the forms. At the lowest level they determine valid inputs and expected outputs for each field in a given transaction and error message descriptions. The model based approach captures this detail as well as the behavior represented by the specifications.

In order to simplify the model creation, generic templates can be created with most of the common information. which are then copied and customized for each transaction.

3.4 Modeling data and maintaining application context within the model

The model can be used to describe flows, but in order to represent the specification and build meaningful tests, data must be represented as well.

An important aspect of testing this AUT is the processing of invalid as well as valid input data. Invalid data can cause hard or soft failures. Hard failure causes no change in the progress state, so the model can repeatedly submit transactions that will cause hard failures enabling the testers to exhaust all of the potential hard errors in a single test case. Soft Failures (i.e. warnings), on the other hand, log a message and change the progress state as if the transaction was successful. In an exhaustive test generation scenario, this may result in many permutations of the ways to successfully progress through all of the progress states. A constraining strategy can be used which allows the testers to configure the model to generate tests that continue or stop after a soft failure. The former was not practical to run during the pilot, the later provided a smaller set of tests that guaranteed coverage of the soft failures, but not the effect of those failures on successive transactions. An example of handling hard and soft failures is shown in Figure 3.

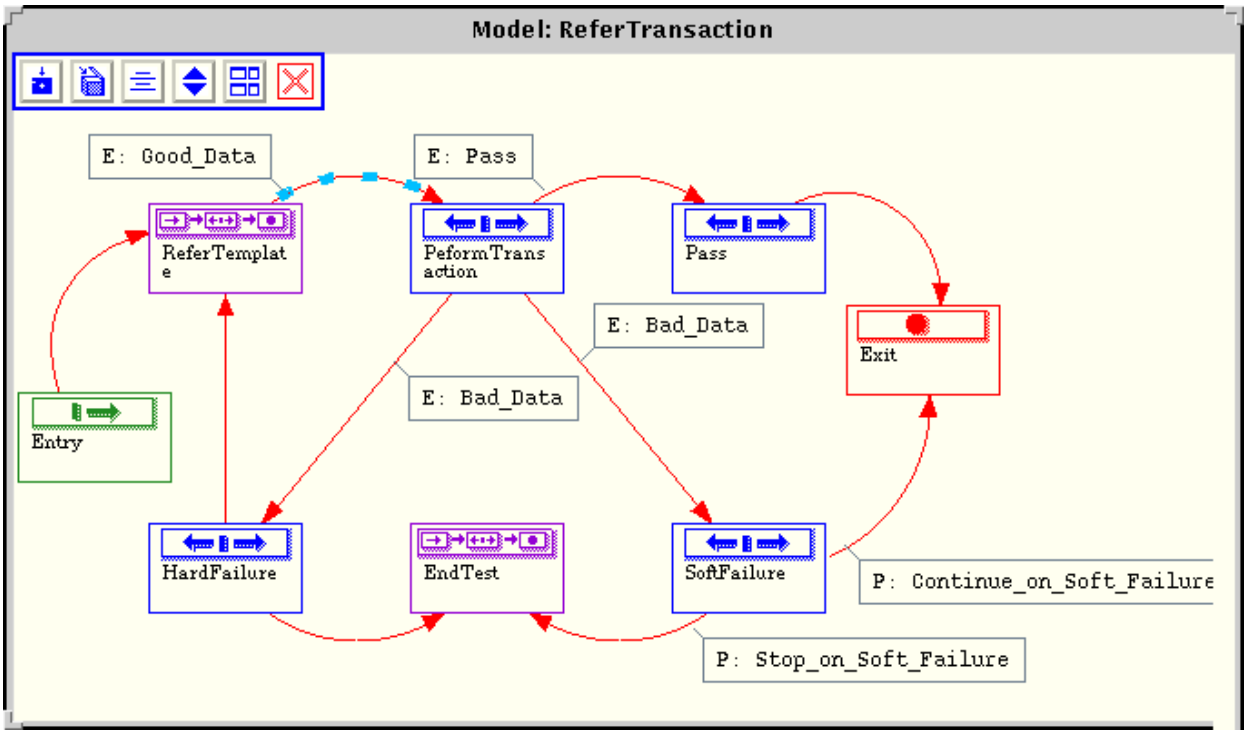


Figure 3: Hard and Soft Failures

Within the model variables are defined whose values can be changed, conditionally checked and output at any point in the flow. The model uses these variables to both maintain context and generate data in the test scripts. Data values are conditionally checked within the model to determine which paths are appropriate in the current context. Multiple values of data are modeled to verify the application behavior over specific domains. For example, a variable called "Transaction" is maintained to represent the state of the "current transaction". When the model is being processed and enters the "CompletedState" state in Figure 2, the value of "Transaction" will be set to "COMPLETE". This action defines the current context for later use in determining the appropriate paths. The string is also used to define input values in the forms, and later verify the DBMS record. The data used to test the application requires more than modeling the values of data. The data fields in a work order are defined in terms of:

- the transaction for which they are valid or common to all transactions
- necessity, whether they are required or optional
- type, input or output
- data type, length and value

Figure 4 shows the model of the forms system for creating a work order. When preparing a work order for the CREATE transaction we have separated the fields of the form into three sub-groups. The fields common to all forms are filled in first (CommonFields), then the fields "required" for CREATE are filled in (CreateReqField), then the "optional" fields are selectively filled in (CreateOptField). Each group is further described in a sub-model, decomposing the system via hierarchy. The sub-models represent individual fields in the form; they contain the detail necessary to generate input data of valid and invalid type, length and value. Optional fields are represented by branched transitions (arrows) with conditional expressions embedded in the transition. An example of a conditional transition occurs in Figure 4, "Transaction==CREATE" (this only allows transactions of type CREATE through this sub-model).

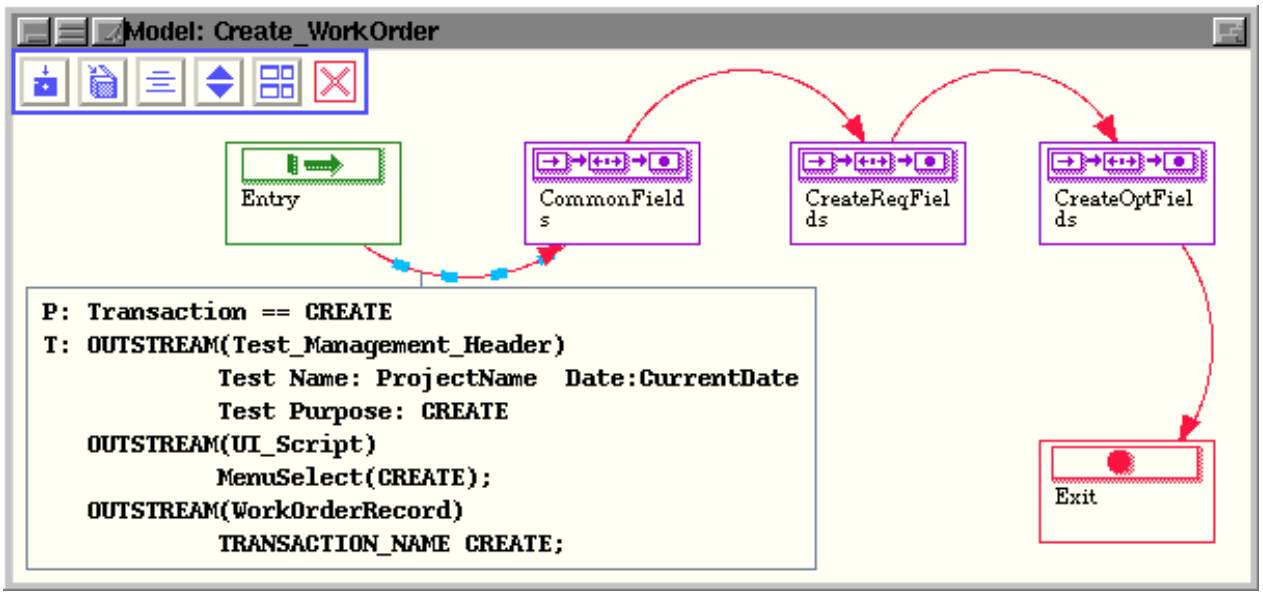


Figure 4: High Level Model of the "CREATE" Form

3.5 Modeling transaction flow to generate database verification records

The Work Order Application is used to create and edit work order records and submit transactions. The model captures the work order process flow, and the values of data throughout this flow. Driving the work order system in a test requires that the data values be output to a forms processing script in the proper sequence. Verification of the transaction processing is more difficult. No feedback is provided through the UI as to the success of the transaction. The scripting facility has no on-screen verification, so reading back the record in a form is not possible. The proper function of the workflow application is verified by querying the database directly with a separate database utility. When a path through the model creates a work order record the verification record is also created as part of the process. For example, if the current state is CREATED, and an UPDATE transaction is submitted, the model contains enough information about the behavior of the system to generate two work order records; a test record is generated for the forms UI, and a verification record for the DBMS. A query is used to confirm that the output record matches the verification record, and if it does not, an error is logged.

3.6 Using constraints to focus tests

The TestMaster system allowed the tests generated to be focused on different classes of user scenarios for the application. Several constraints were defined for the system that enabled a user to generate tests that met specific testing needs. Constraints were used to limit the scope of testing based on: execution time constraints, good and bad test cases as well as specific user functions. The flexibility of the constraints combined with the use of variables in the model itself allowed a wide range of test suites to be generated in minutes.

TestMaster's *Alias* mechanism was used to control the constraints within the model. Conditional expressions were placed throughout the model to constrain the types of scenarios generated. An effort was made to limit trivial cases, cycles, and administrative states. The number of test cases generated for different constraint scenarios are shown in Table 3.

Constraint Description	# of Tests Generated*
Each transaction and each data type in at least one test	20
All permutations of progress state transactions with good data	80
All permutations of progress state transactions with good and bad data*	400
All permutations of progress state transactions and administrative transactions with good and bad data*	3000

Table 2: Number of Test Cases Generated for Different Constraint Scenarios

* Constrained administrative transactions
** All hard failures, stop on soft failure

3.7 Embedding the BUSTER specific information in the model

The test process requires all tests to be archived, logged and run from a central BUSTER test repository (see below). Each test is stored in BUSTER and contains an identifier and test header. The information in the test header is of two classes, basic template information (test name, version number, current date) and test purpose. The purpose of the test is a description of the behavior executed by the test script. Both of these pieces of information are readily available in the model. The model automatically generates the test management system file for each path generated by the model.

4. The Test Management System: BUSTER

This section describes how test cases and the results are stored in BUSTER and how the test cases as generated by TestMaster are imported into the BUSTER Test Storage Hierarchy.

4.1 Test Case Organization

BUSTER requires the test cases to be organized in a directory structure called Test Storage Hierarchy (TSH) as shown in Figure 5. This directory tree can contain multiple levels of directories if it is desired to group lower level functions of each subsystem.

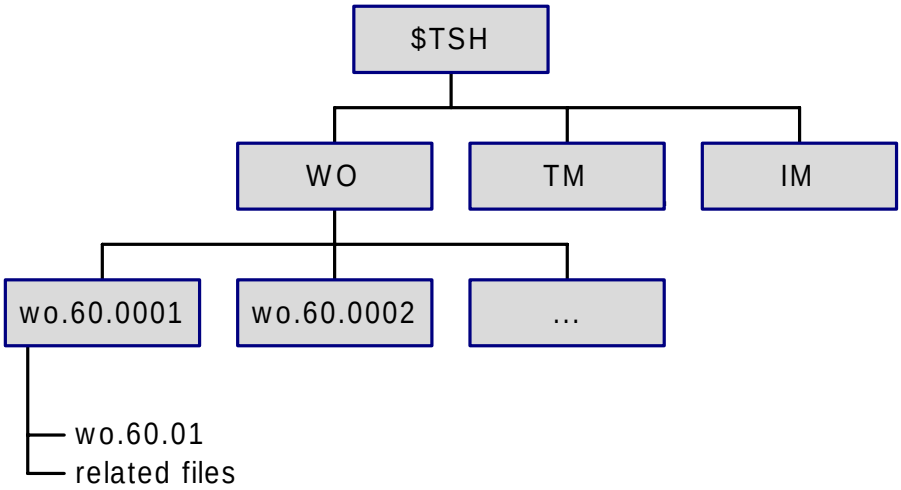


Figure 5: Test Storage Hierarchy

The test case template and all the files related to that test case are stored under the lowest level directory named the same as the test case. Figure 6 shows a sample test case template. The *ID* field is mandatory and is populated when

the test case is created. The other mandatory field is *COUNT* which indicates the number of pass/fail test items within the test case. The *SETUP*, *PROCEDURE*, and *CLEANUP* sections are used for automated test cases and any scripts that can be executed at the SHELL level can be included in these sections. Three BUSTER commands, *pass*, *fail*, and *inc* are used within the test procedure which would record the result of the test case as pass, fail, or inconclusive respectively. *STIME*, *PTIME*, and *CTIME* fields allow testers to limit the execution time of the SETUP, *PROCEDURE*, and *CLEANUP* sections in minutes, in case the lab time is limited or the test script hangs. Any scripts/executables and data files which are used in multiple test cases can be stored in a common directory under BUSTER. The *LIBRARY* field is then used to specify which of these library files are needed during execution, so that when BUSTER packages all the necessary files for execution, these library files are also included in the package.

In addition to the template shown in Figure 6, information about each test case is kept in a database called *testinfo*. The database entry duplicates most of the fields in the test case template and keeps additional information such as the location of the test case and modification date and time. This database allows the usage of SQL to generate reports. More information can be found in Reference [2].

```
ID: wo.60.0001
TYPE: automated
OBJECT:
CONTACT:
DOC:
KEYWORDS:
PACKAGE:
PURPOSE: Create-Pickup-Refer-Confirm-Complete-Verify-Close
REQT:
METHOD:
LIBRARY: verify_progress upd_work_id upd_committime
SHELL: /bin/ksh
SCONFIG:
HCONFIG:
COMMENT:
COUNT: 1
STIME: 5
PTIME: 10
CTIME: 5
SETUP:
PROCEDURE: $TESTROOT/test.sh
CLEANUP:
```

Figure 6: A BUSTER Test Case

4.2 Test Results in BUSTER

When a test case is executed, the BUSTER commands, *pass*, *fail*, and *inc* (inconclusive), used within the *PROCEDURE* section of the template, generates a record of the run session which includes the id of the test case, a unique session number, the date and time of the execution, the result, the total run time, and the time it took to execute the *PROCEDURE* section. This record is then uploaded into the *runinfo* database of BUSTER. The *runinfo* database can be queried using SQL for reports.

4.3 Porting the generated test scripts into BUSTER

TestMaster creates all of the test cases in a big file which is then parsed into the directory structure shown in Figure 7. This directory structure is then moved under BUSTER and the appropriate *testinfo* records are created. *wo.60.0001* is the test case file which has specific information about the test case (Figure 6), *test.sh* is the actual test script that is executed, *get.WO* is the data file which is used to extract the results of each step for verification purposes, and rest of the files are used as input files for each step in executing the test case.

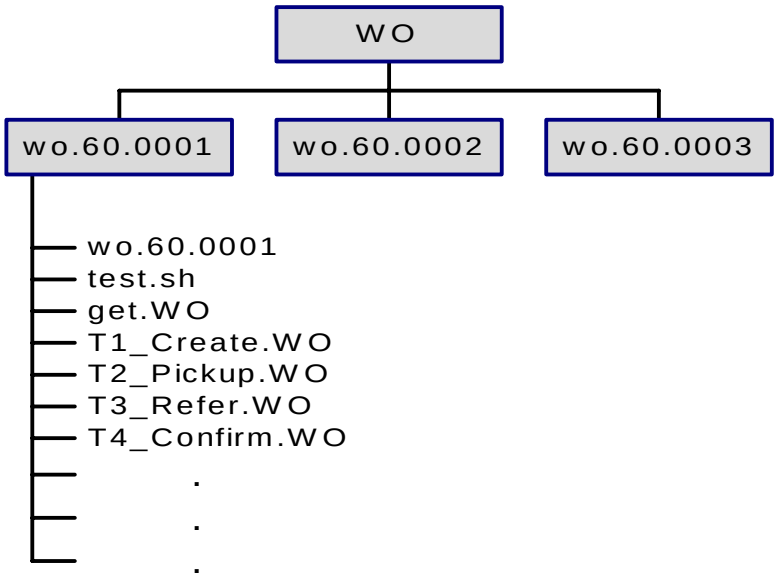


Figure 7: WO Test Storage Hierarchy

Figure 8 shows the contents of *test.sh* for a test case. *TESTROOT* is the environment variable that points to the directory where the test case and all the related files reside during the execution. This variable is set by BUSTER at run time. The script *upd_committime* modifies the Commit Time of the work order as the current time. Once the ticket is created and the WO ID is assigned, *upd_work_id* enters the ID into all the input templates. After each transition, a script called *verify_progress* (listed in Figure 9) is used to examine the output file as well as the database entry. *verify_progress* will issue the *fail* command of BUSTER if the verification fails and will return a non-zero value which will cause *test.sh* to exit. If the verification is successful, the return value will be zero and the next line of *test.sh* will be executed. If all the transitions are successful, the last line will execute the BUSTER *pass* command. This script also issues the *inc* command (for inconclusive) if unexpected results occur, such a wrong command usage and absence of the output file. The scripts *upd_committime*, *upd_work_id*, and *verify_progress* were developed and placed under the BUSTER library, since they are used by almost all of the test cases.

```
$TESTROOT/upd_committime WO || exit
ud < T1_Create.WO > T1_Create.WO.out
$TESTROOT/upd_work_id WO || exit
$TESTROOT/verify_progress WO 0 T1_Create.WO.out CREATED || exit
ud < T2_Pickup.WO > T2_Pickup.WO.out
$TESTROOT/verify_progress WO 0 T2_Pickup.WO.out CREATED || exit
ud < T3_Refer.WO > T3_Refer.WO.out
$TESTROOT/verify_progress WO 0 T3_Refer.WO.out REFERRED || exit
ud < T4_Confirm.WO > T4_Confirm.WO.out
$TESTROOT/verify_progress WO 0 T4_Confirm.WO.out CONFIRMED || exit
ud < T5_Complete.WO > T5_Complete.WO.out
$TESTROOT/verify_progress WO 0 T5_Complete.WO.out COMPLETED || exit
ud < T6_Verify.WO > T6_Verify.WO.out
$TESTROOT/verify_progress WO 0 T6_Verify.WO.out VERIFIED || exit
ud < T7_Close.WO > T7_Close.WO.out
$TESTROOT/verify_progress WO 0 T7_Close.WO.out CLOSED || exit
pass
```

Figure 8: The Test Script *test.sh*

```
##########################################
#  Generic Verify Function
#  Author:  Sadik Esmelioglu
#  Date  :  9/27/96
#  ARG1  :  Object type: WO for work order
#  ARG2  :  Expected Return Code "SR_APPL_RC"
#  ARG3  :  The output file name ex: create.WO.out
#  ARG4  :  Exp prog state "WI_PROGRESSSTATE"
#  Returns  0 If successful
#           1 If Progress state is wrong
#           2 If return code is wrong
#           3 If exact number of args is not supplied
#           4 If output file does not exist
#           5 If get.${OBJ_TYPE} file does not exist
#           6 If get.${OBJ_TYPE}.out file does not exist
##########################################

#
# Initialize the arguments
#
OBJ_TYPE=$1
E_RC=$2
OUT_FL=$3
E_STATE=$4

#
# Check for four input arguments
#
if [ $# != 4 ]
then
  inc -i "$0 is not used properly"
  exit 3
fi

#
# Check for the output file
#
if [ ! -f ${OUT_FL} ]
then
  inc -i "${OUT_FL} does not exist"
  exit 4
fi

#
# Check for the get.${OBJ_TYPE}" file
#
if [ ! -f get.${OBJ_TYPE} ]
then
  inc -i "get.${OBJ_TYPE} does not exist"
  exit 5
fi
```

```
#
# Check the return code
#
A_RC=`grep SR_APPL_RC ${OUT_FL} | cut -f2 -d"
          "`
if [ "${A_RC}" != ${E_RC} ]
then
  fail -f "Expected SR_APPL_RC of ${E_RC} but \
          received ${A_RC}"
  exit 2
fi

#
# Get the object to find out the progress state
#
ud < get.${OBJ_TYPE} > get.${OBJ_TYPE}.out

#
# Check for the get.${OBJ_TYPE}.out" file
#
if [ ! -f get.${OBJ_TYPE}.out ]
then
  inc -i "get.${OBJ_TYPE}.out does not exist"
  exit 6
fi

#
# Check the progress state
#
A_STATE=`grep WI_PROGRESSSTATE \
            get.${OBJ_TYPE}.out | cut -f2 -d"    "`
if [ "${A_STATE}" != ${E_STATE} ]
then
  fail -f "Expected WI_PROGRESSSTATE of\
          ${E_STATE} but received ${A_STATE}"
  exit 1
fi
```

Figure 9: verify_progress Script

The *verify_progress* script, after making sure that all the necessary files are available, it checks the output of the *ud* command for the return code (SR_APPL_RC). If the return code matches the expected return code (second argument supplied to the script), then it "gets" the database record for the order. The script "gets" the database record of the order and checks the output of the "get" operation for the state of the order. If the order state (WI_PROGRESSSTATE) matches the expected state (fourth argument supplied to *verify_progress*), then the script exits successfully.

5. Automated Test Execution

A script, *r_runtest* has been developed to execute a suite of test cases. The following steps are carried out in this script:

1. Create a list of test cases: The script allows the user to specify individual test cases or all the test cases under a directory for execution. Depending on which option is used, the *r_runtest* compiles a list of test cases selected.
2. Extract test cases and library routines: Issuing a BUSTER command, the test cases in the list along with the related files and the library routines are extracted into a package directory.
3. Add BUSTER executables to the package: Since BUSTER resides on a different machine than the application machine, the BUSTER executables and the license file needed on the remote machine are copied into the same directory where the test cases are extracted to.
4. Copy the package to the application machine: The package directory is then remote copied (*rcp*) to the application machine.
5. Remotely start the execution of the test suite: After checking that the remote copy is successfully completed, the BUSTER run command (*brun*) is issued remotely (*rsh*) to start the execution of the test suite.
6. Bring the results file to the BUSTER machine: Following the execution of all the test cases, the r_runtest command, remote copies the results file from the application machine to the BUSTER machine.
7. Upload the results into the BUSTER results database: Once the results file is copied, another BUSTER command (*bstore*) is issued, which uploads the results into the *runinfo* database of BUSTER.

6. Automated Reporting

6.1 Reports on demand

Testers access BUSTER on a daily basis for planning and tracking the testing activity. They are interested in finding out how many test cases remain to be tested so that a decision can be made whether there is sufficient time to execute them or not. It is also necessary to find out which test cases have failed so that so that proper defect reports can be written and also re-tested when defects are fixed

Scripts are developed to extract and display this type of information from the BUSTER databases. Below is a list of these scripts and a brief explanation of what they do:

- **testlist <path>** provides a list of test cases under a path
- **oneliner <path>** provides a list of test cases and their purposes under a path
- **tstres <path>** reports the number of tests written, run, passed, and failed under a path
- **tststat <path>** displays results of each execution for the test cases under a path
- **failed <path>** provides a list of failed test cases under a path
- **notrun <path>** provides a list of test cases under a path which are not run

6.2 Weekly reports

Weekly reports are generated for project management and show the progress of testing efforts by major features. These report are prepared in *troff*, since it is easier to automate the generation of the tables and graphs and the data from BUSTER that go into the report.

The weekly report has three sections:

1. Cover Page which is a troff template updated manually by the test coordinator and includes verbal information such as the load that is being tested and problems encountered.
2. Test Table which shows number of test cases scheduled, executed, and passed for each feature.
3. Graphs for each feature showing number of test cases scheduled, executed, and passed by week.

In order to facilitate the automatic generation of the weekly report, another database is created under BUSTER called *grapinfo*. This database contains weekly execution plan for each feature and is populated prior to test start date.

Every week during the testing period, the test coordinator manually edits the report cover page with any verbatim information and runs the report generator which goes to the databases testinfo, runinfo, and grapinfo to generate the test table and graphs in *troff* format.

6.3 Reports on the Web

On line reporting of the testing status is a major convenience for the test coordinator who has to distribute the information to multiple people and sometimes in multiple locations. It is also more convenient to the audience of the reports to access it electronically whenever they need and from wherever they are. The intranet Web provides an excellent platform for on line reports.

Both the weekly and on-demand reports can be displayed on the Web. Weekly reports are converted into the postscript format and accessed through a Web browser via a postscript viewer. Most commonly used postscript viewers are Ghostview (shareware and available both for PC and UNIX platforms) and pageview (on SunOS platforms). The on-demand reports are basically URLs pointing to cgi shell scripts which execute the report commands listed in Section 6.1 and put the HTML wrappers.

Figure 10 shows the *index.html* file for the main report web page. This page points to the weekly reports page, *weekly.html* (Figure 11), and the cgi shell script, *get_feats.cgi* (Figure 12), which allows the users select a project and displays the page for on-demand reports.

The main page can be expanded to include other projects and/or other releases easily, by duplicating the “Release” specific lines and modifying the release or project names.

Every week, the test coordinator copies the postscript report file into the directory where the *weekly.html* file resides and modifies the file to add a line for that week. This task can also be automated and included in a cron job which gets executed automatically on a weekly basis.

The script, *get_feats.cgi*, goes to the BUSTER machine and finds the names of the top-level directories for the selected project, then displays a form to select one of the directories and a list of commands to choose from. Once the user selects a directory name (or ALL) and a command, this script executes another script passing the directory, command, and the project names as arguments. This script, *test_rep.cgi* (which is not provided in this paper for brevity), then goes to the BUSTER machine, executes the selected command, wraps the results with HTML commands, and displays it.

```
<TITLE> System Test Reports</TITLE>
<BODY BGCOLOR="#00DDAA">
<CENTER><H1>Project System Test Reports</H1></CENTER>

<H2> <A HREF="weekly/weekly.html">Release 2.0 Weekly System Test Reports</A></H2>
<H2>Test Reports:</H2>
<FORM METHOD="GET" ACTION="scripts/get_feats.cgi">
Project:
<SELECT NAME="project">
  <OPTION> Release1.0
  <OPTION SELECTED> Release 2.0
</SELECT>
<INPUT TYPE="submit" VALUE="Select Project">
</FORM>
<HR>

<ADDRESS><B>sadik@lucent.com</B></ADDRESS>
</BODY>
```

Figure 10: Report Main Page (index.html)

```
<TITLE>Weekly Test Results</TITLE>
<BODY BGCOLOR="#00DDAA">
<CENTER><H1>Release 2.0 Weekly Test Reports</H1></CENTER>

<H2>Test Status (Postscript)</H2>
<BL>
<LI><A href="report2.ps">Week 2 Ending 03/30/97</A>
<LI><A href="report1.ps">Week 1 Ending 03/23/97</A>
</BL>
<HR>

<ADDRESS><B>sadik@lucent.com</B></ADDRESS>
</BODY>
```

Figure 11: weekly.html

```
#!/bin/ksh
# Send the required HTML header
cat <<-EOM
        Content-type:  text/html

        <TITLE>Test Reports</TITLE>
        EOM
# name=value pairs are passed via QUERY_STRING env variable the following eval parses the
# name=value as shell env variables;  sed cleans out quotes and awk does the parsing
# as well checking for illegal shell variable names
eval `echo $PREFIX$QUERY_STRING | sed -e 's/"""/%27/g' | \
  awk '
     BEGIN{RS="&";FS="="}
     $1~/^[a-zA-Z][a-zA-Z0-9_]*$ {printf "QS_%s=%c%s%c\n",$1,39,$2,39}' `
# Form a list of features
print "ALL" > /tmp/junk$$
rsh buster_mach -l user ". /buster/.busterenv $project >/dev/null;ls \$TSH" >> /tmp/junk$$
cat <<-EOM
        <CENTER><H1>Test Reports For $project</H1></CENTER>
        <FORM METHOD="GET" ACTION="test_rep.cgi">
        <INPUT TYPE=hidden NAME="project" VALUE="$project">
        <BL>
        <LI><B>First Select a Feature</B>
        <BR>
        <SELECT NAME="feat">
        EOM
while read feat
do
  print "  <OPTION> ${feat}"
done < /tmp/junk$$
print "</SELECT>"
cat <<-EOM
        <LI><B>Then Select One of the Following Reports on that Feature</B>
        </BL>
        <TABLE>
        <TR>
        <TD><INPUT TYPE=submit NAME="request" VALUE="Listing"></TD>
        <TD>List of Test Cases</TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=submit NAME="request" VALUE="Statistics"></TD>
        <TD>Test Statistics</TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=submit NAME="request" VALUE="Results"></TD>
        <TD>Individual Test Case Results</TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=submit NAME="request" VALUE="Notrun"></TD>
        <TD>List of Test Cases that are Not Run</TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=submit NAME="request" VALUE="Failed"></TD>
        <TD>List of Test Cases that Failed</TD>
        </TR>
        </TABLE>
        </FORM>
        EOM
rm /tmp/junk$$
```

Figure 12: get_feats.cgi Script

7. Conclusions

The goal of this pilot project was to shorten the regression test cycle time and reduce the effort needed. Three areas were targeted to reach the goal: (1) Test Case Generation, (2) Test Execution, and (3) Reporting. By automating these three phases major savings have been achieved, which are summarized in Table 3.

	Manual	w/TestMaster & BUSTER
Pilot Model	NA	10 person-days
Model Change	NA	15 min
Test Case Generation	60 min/tc	3 sec/tc
Test Case Execution	3-10 min/tc	2 min/tc
Test Case Reporting	3 min/tc	0 (done w/execution)

Table 3: Savings Achieved

TestMaster requires a functional model to be built which took 10 person-days. After this initial investment, most of the subsequent changes to the model are achieved within 15 minutes. Having a model of the features not only facilitates the test case generation, but also visually demonstrates the functionality to project members and serves as an excellent learning tool for new-comers.

Test Case Generation involves listing of possible scenarios for testing and coming up with the necessary input files and test scripts for execution. The current methodology required creating these files manually which took 60 minutes per test case on the average and was prone to mistakes. Generating test cases with TestMaster took an average of three seconds per test case. The savings in execution was not as much, but analyzing the results automatically added to the savings achieved and was more accurate. Automated reporting, on the other hand, was part of execution as opposed to the current manual process and did not require the time needed to enter the results into BUSTER and manually create reports. The savings for the pilot which automated the generation, execution, and reporting of one feature of one release was about **60 minutes** per test case. The pilot generated 49 test cases with maximum constraints and had to be stopped after generating thousands of test cases with no constraints. Relaxing the constraints to increase the coverage, 80 test cases can easily be generated which recovers the initial cost of 10 person-days during the regression testing of the first release.

8. Acknowledgments

The authors wish to thank Chrysanthi Kefala, Debbie Kao, and Peter Hartgrove of Lucent Technologies for providing extensive support, Chrysanthi and Debbie with their product expertise and Peter with his knowledge of the test environment. We would also like to thank JD Doyle of Teradyne for his modeling support and the development of the parsing script.

9. References

[1] Harry Robinson, “An Introduction to ORBIT Testing”, AT&T Internal Memorandum, August 1995.

[2] “BUSTER Test Management System - User’s Guide”, Issue 5, September 1994.

[3] S. Esmelioglu, “BUSTER+: A Test Support Tool”, AT&T Internal Memorandum, Issue - 2, September 1992

[4] S. Esmelioglu, “Test Status Reports Using BUSTER+”, AT&T Internal Memorandum, January 1993

[5] L. Apfelbaum, “Automated Functional Test Generation”, Proceedings of the Autotestcon ’95 Conference, IEEE, 1995.

[6] L. Apfelbaum & J. Doyle, “Model Based Testing”, Proceedings of the Software Quality Week ’97 Conference, IEEE, 1997.

Parlez-Vous Klingon?
Testing Internationalized Software with Artificial Locales

Harry Robinson, Arne Thormodsen

Hewlett Packard
Workstation Technology Center
Corvallis, OR

Introduction

yIvoq ’ach yI’ol
("*Trust, but Verify.*") [1]

Many companies would like their software to run in multiple languages so that they can market it around the world. **I18N** [2] is a software methodology that makes it possible to create internationalized software at a reasonable cost by separating the executable code from any user interface components. To adapt I18N software to a new language, only the user interface needs to be translated; the executable code remains unchanged.

Writing I18N software can be straightforward. However, there are many linguistic, technical and psychological obstacles to performing good testing on that software. To exercise I18N code thoroughly, translated user messages are needed, but these translations are typically not available until late in the development cycle. And if the testers do not understand the language, the translated text may be intimidating and frustrating to use.

To overcome these obstacles, we have created user interface artificial locales such as Klingon(tm) and Swedish Chef(tm) for testing I18N code. This paper explains the reasoning behind the locales and how they provide better testing of our software early in the development lifecycle. A specific application of these techniques to the generation of UNIX message catalogs is described.

Writing Internationalized Code

qo'mey poSmoH Hol
("*Language opens worlds.*") [3]

In UNIX-based internationalization, strings that are to be displayed to the user are separated from the executable code and placed into files called message catalogs. Each string in the message catalog is indexed by a set and message number, and the I18N code accesses a string by this index. (Some non-UNIX systems use *application resource files* instead of message catalogs; the only difference is that application resources are accessed by the resource name rather than a numeric index.)

Instead of having hardcoded strings in a program such as

```
printf("hello, world\n");
```

I18N stores the string in a message catalog as follows:

```
$set 1
5 hello, world\n
```

The code retrieves this string by accessing message number 5 in message set 1. This arrangement is very useful for translations. To translate our "hello, world" program into French, a translator merely changes the message string to be

```
$set 1
5 bonjour, le monde\n
```

No changes need to be made to the I18N code to support the French version of "hello, world".

As a real-world example, the English and Japanese Text Editor Help Menus in Figure 1 below were both generated by the same software; only the message catalog was changed.

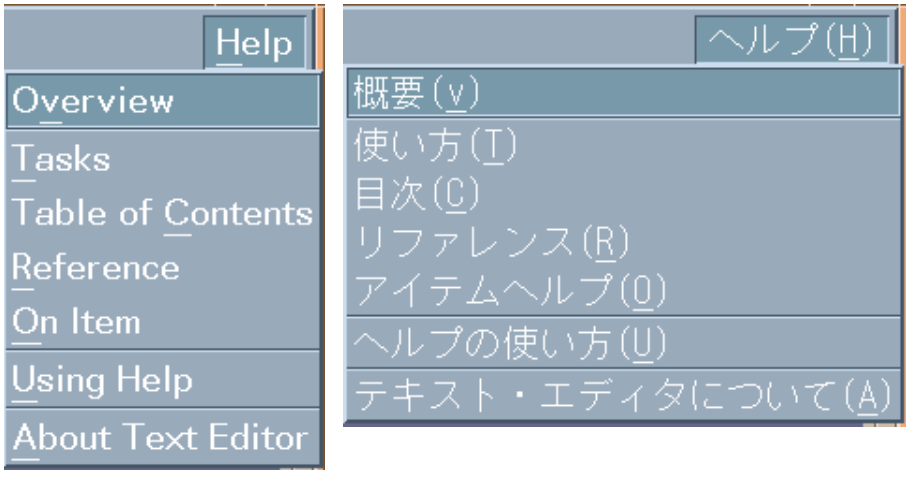


Figure 1: English and Japanese Help Menus

Problems with Testing Internationalized Code

Hoch ’ebmey tIjon
("*Capture all opportunities.*")

Here is the typical process for developing internationalized applications:

1. the development team writes an application using the default English locale
2. the development team distributes the English locale message catalog to localizers
3. the localizers translate the English locale messages into other languages
4. the development team receives message catalogs back from the localizers
5. the development team integrates and tests the application with the new message catalogs.
6. the localizers test the localized application to verify the translation.

Since message translation (step 3) is a lengthy procedure, the development team may have to wait several weeks before testing the internationalized parts of their code. Precious test and debugging time is lost waiting for the translated catalog to come back from the localizers.

There are several common mistakes people make when writing I18N code. One mistake is to neglect to leave enough room in the message buffer for the translated message string. Some languages, such as German, require more space than English does for the same message. The length of the translated message string cannot be known until runtime, though a good rule of thumb is to allow for 60% text growth during translation. If the translated message string is still too long for the application to handle, the application should deal with it gracefully, for instance, by truncating the translated string to a reasonable length.

A second common mistake is when developers neglect to accommodate languages, such as Japanese, that require two bytes to store a single character. Most Western languages require only one byte of storage per character, but several Far Eastern languages have large characters sets and need more than one byte per character. If the code does not handle double-byte characters gracefully, the results could range from corrupted characters to a crash of the application.

Testing for these kinds of errors is difficult because translated message catalogs are necessary. As noted above, it may be many weeks before an actual translated message catalog is available. Even when the message catalogs are available, many people, lacking familiarity in the target languages, are reluctant to test the application in locales such as Japanese which are most likely to expose problems.

Also, there is no guarantee that the strings in an official catalog will expose defects in the code. If, for instance, the code does not handle long strings gracefully, we will not see that defect unless the message catalog we are using has very long strings.

We need a fast, simple way to generate message strings that are easy to use in testing and are likely to expose defects.

Our solution has been to construct artificial language message strings that mimic the kinds of problems we see in actual message strings. This instant creation of a translated message string against which to test our software provides us with quick feedback about how our application will perform with actual localized components.

Creating an Artificial Locale Message Catalog

tlhutlhmeH HIq ngeb qaq law' bII qaq puS
("*Drinking fake ale is better than drinking water.*")

Translating the messages in a message catalog is more difficult than translating plain text, regardless of whether it is a human or a machine doing the translation. The problem lies in the fact that a message catalog, in addition to having words that must be translated, also has a very definite structure that must be followed.

- At the very simplest level, all strings in a message catalog are indexed by a set number and a message number. Any valid translation of a message catalog must include all the same set and message numbers as the original.

- Many applications use menu mnemonics. To preserve the functionality of the menus, these mnemonics should be kept as single ASCII characters associated with a menu label.

- Format conversion specifiers such as "%s" or "%d" in the message strings must be copied into the new catalog, unchanged except for the possible addition of positional parameters. [4]

- Control characters such as "\n" and "\t" must be copied into the new catalog unchanged.

- All other characters in the message string can be "translated" in any way we deem useful.

Our approach to creating a new message catalog is straightforward. We pass each English message string through a filter that separates the characters we wish to translate from those, such as control characters and format specifiers, that we do not. We then run one of several filters on the characters we wish to translate. We then construct the new message string by merging the newly translated characters with the untranslated characters from the earlier string.

The Swedish Chef Locale

Dal pagh jagh
("*No enemy is boring.*")

To verify that the software handles long strings correctly, we create a locale where the strings are all significantly longer than their English counterparts. A simple implementation might be to append a static string, such as the alphabet, to each string in the catalog. Preserving the original string in this way makes it easy for the person looking at the screen to determine what the original string said.

We chose a slightly more whimsical approach. We found a program called "Encheferizer" [5] on the Internet that converts English words into a semblance of the speech patterns of the Swedish Chef(tm) from the classic Muppet(tm) television show. (To ensure that the "translated" strings would all be long, we modified the Encheferizer code slightly to append "Bork! Bork! Bork!" to each string.)

Here is the Swedish Chef version of the Help Menu:

Figure 2: Swedish Chef Help Menu

As you can see, the modified "Encheferizer" expands the size of the strings in this menu to roughly double their original size. Other "translation" filters can be substituted if a greater effect is desired.

The Wide Locale

mataHmeH maSachnIS
("*To survive, we must expand.*")

A common place for programs to fail is in handling codesets that require more than one byte to represent a character. Japanese SJIS (pronounced "Shift-JIS"), for instance, has a very large character set and requires two bytes to store each character. Double-byte characters often cause trouble for an I18N program, and it is useful to verify handling of double-byte characters early in development.

There are two main problems with using the actual Japanese SJIS locale to test the handling of double-byte characters. First, the translation may take weeks. And second, even after the translation is completed, most testers don't feel comfortable running the application in Japanese.

We answer these problems by filtering the strings we wish to translate through a small C language program that maps ASCII characters into recognizable double-byte counterparts that are intelligible to English readers.

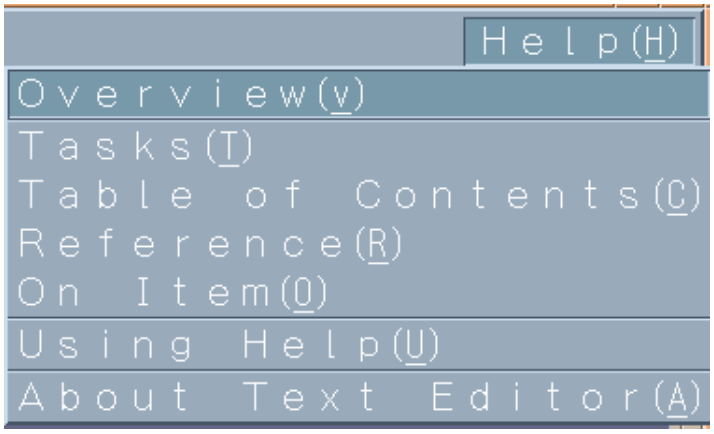


Figure 3: Wide C Help Menu

It is important to note that the strings in the menu, such as "O v e r v i e w" are not made up of ASCII characters; rather, they are composed of double-byte characters that look like their ASCII counterparts. (For accuracy's sake, the actual characters are JIS 0208 Latin, using the SJIS encoding.) The actual representation of the string "O v e r v i e w (v)" in the message catalog is

\202n\202\226\202\205\202\222\202\226\202\211\202\205\202\227(v)

The Wide locale is useful for finding places where the code does not handle double-byte characters. It is also useful for locating strings that are hardcoded into the software. Since we have translated all the strings in the message catalog into double-byte format, any strings that look "thin" must be coming from somewhere else. The most common source for these strings is that they are hardcoded into the software, or that the software is failing to access the message catalog correctly.

The Troublesome Wide Locale

bIQapqu’meH tar DaSop ’e’ DatIvnIS
("*To really succeed, you must enjoy eating poison.*")

A further twist on the Wide Locale scheme is to prepend to each label a string of double-byte characters which are known to be difficult to process because their second bytes correspond to ASCII delimiter characters such as backslash (’\’) or double quote (’"’). These types of characters are present in only a few codesets, such as ja_JP.SJIS, but can be the source of much difficulty. Typically, they will be misinterpreted if the system is not configured properly to interpret them, causing undetected data corruption.

Here is an example of a menu with such a string prepended to each entry. The string is a list of Japanese "katakana" characters which were selected to be troublesome. In this case, the strings are displayed correctly because the system was configured correctly when the menu was compiled. When the same menu was built on an incorrectly configured system, the data corruption was so severe that the application would not even start.

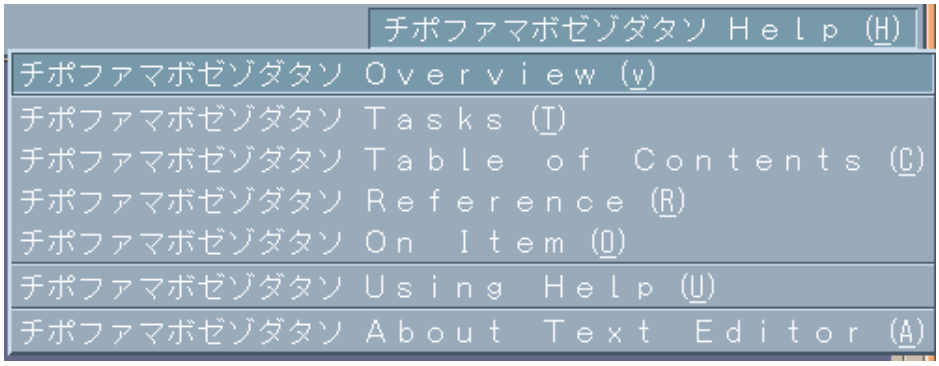


Figure 4: Troublesome Wide C Help Menu

The "troublesome" type of test is useful for finding a class of problems that are not common, but quite difficult to detect by other means. It is not as convenient as the other example because non-English characters are involved. However, it is also generally applied only once, to one component, to reveal if there are system-wide configuration problems.

The Klingon Locale

tlhIngan Hol Dajatlh'a'
("*Do you speak Klingon?*")

One persistent complaint we hear is that people feel that they cannot test in a language that they do not understand. Since our test approach claims that it is possible to construct many tests that are "locale-independent" [6], we decided to prove the claim by choosing a language that would be equally difficult for almost anyone in the world.

Having found the Klingon version of ***Hamlet*** [7] on the Internet, it was simple work to construct a word list for the Klingon language. The Klingon filter substitutes random Klingon words for the sections of the message that we wish to translate. To make the test even tougher, we convert the single byte characters into double-byte characters as we did with the "Wide C" locale. (We also investigated using the official Klingon *Klinzhai* characters, but no version of the fonts was available for UNIX.)

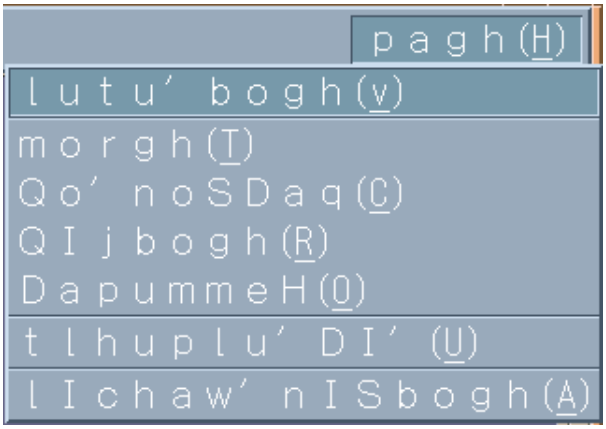


Figure 5: Klingon Help Menu

This "random word" replacement method was easy to implement, and it avoided the awkward issue of whether Klingons would even provide a Help menu in their applications.

Advantages of Artificial Locales

reH Suvrup tlhIngan SuvwI'
("*A warrior is always prepared to fight.*")

Artificial locale message catalogs are useful in early detection of the type of bugs that appear in internationalized software. It eliminates serious technical and psychological barriers to adequate testing of I18N applications by providing messages that are easy to use and useful at exposing bugs.

Disadvantages of Artificial Locales

Dujeychugh jagh nIv yItuHQo’
("*There is nothing shameful in falling before a superior enemy.*")

The "translation" filters mentioned in this paper are general-purpose utilities, and do **not** take into account special cases of strings that should not be translated. An example of such a string would be a single-byte string that gets sent to a printer interface. Some special arrangement would have to be made to preserve that string’s functionality.

These are not real translations, and testing in these locales does not eliminate the need for testing with the actual message strings when they become available. There are always bugs that crop up in the real world that are difficult or impossible to predict with a simulation.

Conclusion

Qapla’
("*Success!*")

Actual translations of message catalogs arrive too late in the development process to be useful in eradicating many internationalization bugs. Since we can anticipate some common bugs in I18N code, it is a good idea to create artificial message strings so that we can test for and eliminate these bugs as soon as possible. It is an added benefit to make the "translated" message strings as useful, simple and pleasant to work with as possible.

References

reH tay’ ghot tuqDaj je
("*One is always of his tribe.*")

[1] Marc Okrand, **The Klingon Way: A Warrior’s Guide**, Pocket Books, 1996 (Unless otherwise noted, all Klingon quotations come from this source.)

[2] An excellent introduction to the I18N methodology is Thomas McFarland’s **X Windows on the World**, Prentice-Hall, Inc, 1996

[3] Motto of the Klingon Language Institute, http://www.kli.org/

[4] Positional parameters are an extension of the format conversion specifier that allows localizers to

change the order in which arguments appear in the output. Positional parameters are discussed in [2], pp 67-69

[5] John Hagerman, "*Ze sveedish chef*", http://www.almac.co.uk/chef/chef/chef.html

[6] Harry Robinson and Sankar L. Chakrabarti, *Testing CDE In Sixty Languages: One Test Is All It Takes*, Proceedings of the 14th International Conference on Testing Computer Software, 1997

[7] Nick Nicholas & Andrew Strader, **Hamlet, Prince of Denmark (The Restored Klingon Version)**, 1996, ftp://ftp.kli.org/pub/Text/KSRP/hamlet

---

Harry Robinson (harryr@cv.hp.com) and Arne Thormodsen (arnet@cv.hp.com) are R&D engineers at the Workstation Technology Center, Hewlett Packard Co., 1000 NE Circle Blvd., Corvallis, OR 97330.

View Slides

Guerrilla SQA

Dave Duchesneau, *The Boeing Company*

Abstract

Business is war, and quality is a battle-ground! The military metaphor provides rich analogies for reasoning about how to optimize QA. Learn how to leverage your resources to find bugs earlier (reducing development cost) and slash failures in the field (reducing maintenance and operational costs). Learn about five overarching fundamentals and seven underlying principles to keep from sabo-taging your own efforts.

Dave Duchesneau *is a computer scientist and senior quality analyst for Boeing. He has been programming since 1973 and specializes in OO technology and prac-tices. Dave has presented at several international conferences and published dozens of technical articles.*

Simulating Specification Errors and Ambiguities in Systems Employing Design Diversity

Jeffrey Voas
Reliable Software Technologies
Suite 250
Research Division
21515 Ridgetop Circle
Sterling, VA 20166
jmvoas@RSTcorp.com

Lora Kassab*
Naval Research Laboratory
Center for High Assurance Computer Systems
Code 5542, 4555 Overlook Avenue SW
Washington D.C 20375
kassab@itd.nrl.navy.mil

Abstract

This paper looks at methods for predicting how likely it is that an *n*-version software system will suffer from common-mode failures. Common-mode failures are frequently caused by specification errors, specification ambiguities, and programmer faults. Since common-mode failures are detrimental to *n*-version systems, we have developed a method and a tool that observes the impact of simulated specification errors and specification ambiguities. These observations are made possible by a new family of fault injection algorithms designed to simulate specification anomalies. As a side-benefit, this analysis also provides clues concerning which portions of the specification, if even slightly wrong or misinterpreted, will lead to identical failures by two or more versions. This suggests which specification directives have the most impact on the system's functionality.

Keywords

n-version programming, fault injection, common-mode failure, specification

Biographies

Jeffrey Voas is a Co-founder and Chief Scientist of Reliable Software Technologies. Voas has coauthored a text Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995). Voas is currently co-authoring a second text "Software fault-injection: inoculating programs against errors", due to be published by Wiley in 1997.
Lora Kassab graduated from the College of William and Mary in May 1997 with an MS in computer science. She is a computer scientist in the Information Technology Division at the Naval Research Laboratory in Washington D.C. Her interests are computer security, testing, and distributed systems.

*This work was performed when Kassab was a graduate student at the College of William & Mary.

1 Introduction

Software systems suffer from a variety of problems: incorrect requirements and specifications, programmer faults, and faulty input data. These problems can cause software to exhibit undesirable behavior, including crashing, hanging, or simply just producing wrong output.

At best, *software testing* reduces programming faults, but software testing can do little for specification errors or corrupt input data anomalies. *Formal methods* are geared toward thwarting large classes of inconsistencies and ambiguities in specifications and can even detect programmer faults. But even formal methods can be misapplied or fail to detect specification errors.

Here, we will provide a set of algorithms that are similar to traditional software testing approaches, but instead of testing the software, they test the resilience of the software to specification errors. New software fault injection algorithms will be introduced here. These algorithms provide insight into how a system will behave if the specification is erroneous.

Previously, we have published results from using fault injection in applications that did not employ design diversity [9]. In this paper, we are focusing on ways for predicting whether the identical failure by two or more parallel versions is possible if the versions' faults can be traced back to a common specification error or specification ambiguity. More specifically, we are interested in how identical failures by *diverse* versions affects an *n*-version system. (We have also done similar work that deals exclusively with using fault injection to simulate random anomalies in diverse versions [10], but that is not our focus here.)

In our approach, each version is forced to experience an incorrect internal computation that can be mapped back to a common specification error. By showing that common-mode failures are infrequent after the specification is forcefully mutated in a manner that simulates specification errors and ambiguities, we can plausibly argue for placing diverse software versions in parallel (with the use of a voter).

In *n-version programming*, different software versions, written to the same specification but developed independently, execute in parallel (See Figure 1). It is imperative that there is *no* communication between the teams responsible for developing the different versions. Quarantining the different teams is essential such that misunderstandings from one team do not affect the understanding of other teams. But quarantining teams is not always enough–uncorrelated faults in distinct versions can lead to identical failures.

Although design diversity thwarts certain types of faults, it does not thwart all faults [6, 2]. Knight and Leveson [6] demonstrated that different programmers can make the same logical error. An additional result from Knight and Leveson demonstrated cases where different logical errors yielded common-mode failures in completely distinct algorithms and in different parts of similar algorithms.

For this paper, we need to differentiate three different types of failures: (1) failures of versions that satisfy the definition for common-mode failure, (2) non-common–mode version failures, and (3) voter failures. *Common-mode failure* occurs when two or more identical software versions are affected by faults in exactly the same way [4]. More specifically, common-mode failures are said to occur when there exists at least one input combination for which the outputs of two or more versions are erroneous, and all outputs are identical for all possible inputs for this combination of versions. Thus, if two or more versions respond to all inputs in the same way, and there is at least one input that causes this set of versions to fail, then common-mode failure has occurred. *Non-common–mode failures* are simply version failures that cannot be correlated with other versions

and thus do not satisfy the definition of common-mode failure. *Voter failures* occur when the voter makes a wrong decision because of the inputs it received.[1]

It is reasonable to further classify the severity of common-mode failures, since certain classes of common-mode failures are more likely to trip up the voter than others. Here, *severity* is equal to the number of versions that are in agreement on the "wrong output." For example, a common-mode failure between two versions is less likely to result in voter failure for a 9-version system than for a 3-version system. This explains why specification ambiguities that affect only a handful of the versions can be benign if the system is sufficiently large (in terms of the number of versions).

Many people have written off *n*-version programming as a dead approach to attaining high integrity software because of the *n*-version problem. But to our amazement, *n*-version programming is alive and well in several different safety-critical domains, and it is particularly popular outside of the United States. For example, Airbus uses diverse software versions for the A320/A330/A340 electrical flight controls systems [7, 1]. (But *n*-version programming cannot fix everything—the *n*-version Airbus flight control system still has a fatal flaw that has not been fixed or thwarted by any of the *n*-version capabilities [5].)

In the United States, the FAA's position, based on industry's feedback, is that since the degree of protection afforded by design diversity is not quantifiable, employing diversity will only be counted as additional protection beyond the prescribed levels of assurance but will not substitute for other requirements [3]:

> The degree of dissimilarity and hence the degree of protection is not usually measurable. Probability of loss of system function will increase to the extent that the safety monitoring associated with dissimilar software versions detects actual errors or experiences transients that exceed comparator threshold limits. Multiple software versions are usually used, therefore, as a means of providing additional protection after the software verification process objectives for the software level have been satisfied.

2 Software Fault-injection

A significant amount of research has focused on methods to detect and eliminate errors earlier in the software life-cycle *e.g.*, prior to implementation. Even so, errors related to misunderstandings, ambiguities, or faulty assumptions will find their way into specifications. This is inevitable.

Many people have spent careers trying to develop techniques that eliminate all program errors. (Although laudable, the fact remains that not all errors need elimination: only those errors that have nasty consequences.) Because certain errors can be tolerated, we wish to isolate those classes of specification errors that if implemented in an *n*-version system, will cause the voter to make a bad choice. For now, a "bad" choice will be the same as a "different" choice (i.e., the specification error had not been programmed).[2] By demonstrating that particular classes of specification errors and ambiguities are unlikely to impact the voter's decision, increased confidence is warranted for employing redundant, diverse versions in parallel.

[1] We will ignore the possibility of faulty voters.

[2] Because we have built other utilities to detect different types of internal states and output events, "bad" could also be defined as other failure classes (such as "unsafe").

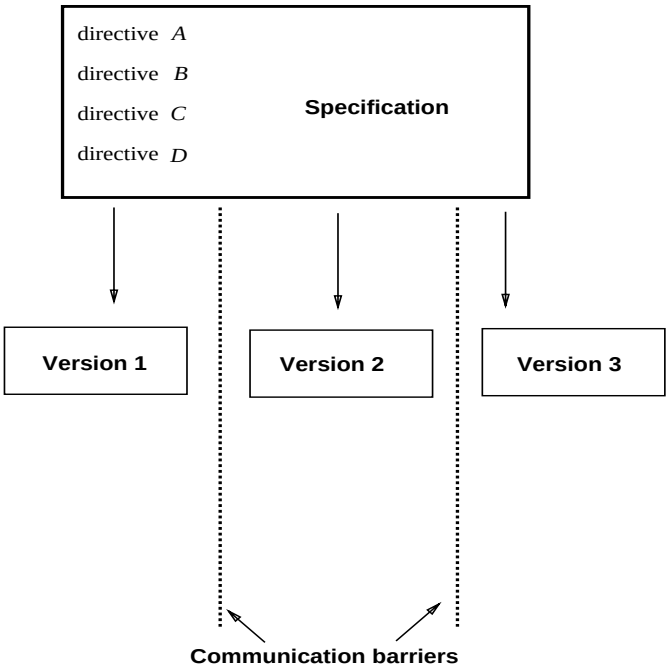


Figure 1: One Specification and Three Independent Versions.

Let's begin by considering the simple *n*-version system in Figure 2. This figure illustrates the traditional architecture of an *n*-version system that is composed of *n* independent versions and a software voter *V*. *i* is the input value fed to each version in parallel. *V* determines which output to release from the *n* versions. We employ software fault injection to determine whether identical programmer faults in two or more versions will cause identical version failures. If so, then we know that the voter will also succumb to the identical programmer faults. And if the faults have a common root cause such as a faulty specification, then we know which classes of specification errors must not occur.

The process of performing software fault injection involves adding code to the code under analysis; the added code is called *instrumentation.* The modified program is then compiled and executed. The instrumentation is involved in either injecting anomalies or observing the impact of the anomalies.

There are many different types of specification-based anomalies that could be simulated using fault injection. The key classes of specification-based anomalies that should be simulated via fault injection methods are:

1. Those anomalies that can arise from actual *specification errors*, where if each programmer implements the specification correctly, then each version will perform some internal computation differently. (Whether this forces the versions to produce an incorrect output is another question.)

2. Those anomalies that can arise from *specification ambiguities*, where at least one programmer implements some directive $\mathcal{C}$ in manner $\mathcal{C}_a$, and the remaining development teams implement it in a semantically different manner, $\mathcal{C}_b$. (Again, whether this forces the versions to produce an incorrect output is another question.)

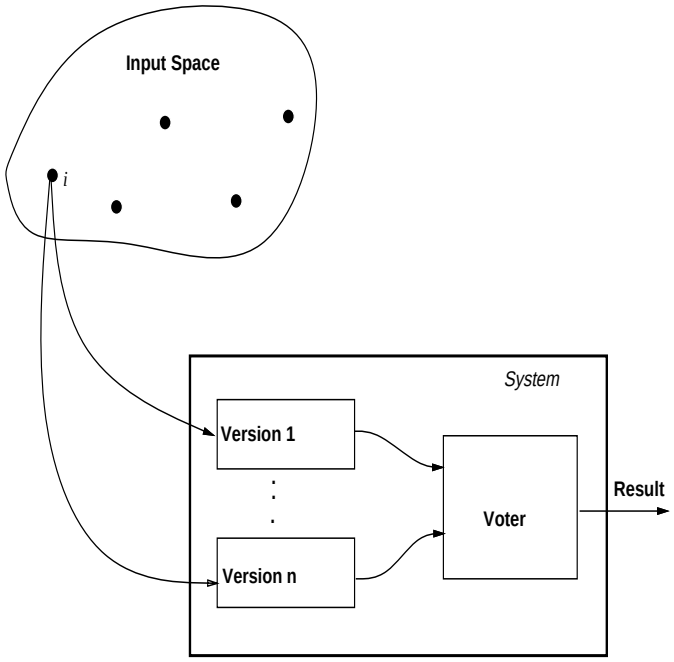


Figure 2: *n*-version System with Voter and Inputs.

2.1 Analysis Assumptions and Requirements

This approach is a *behavioral analysis* technique. The appropriate time to apply this technique is after the *n*-version system is built and ready for deployment. This approach assumes that the following are available:

1. Version outputs can be captured before they enter into the voter.
2. Source code to the versions is available.
3. A specification *S* from which we can isolate specific directives, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ...

In addition, the approach does not require the following be true, but it is prudent that these assumptions are also true.

1. The versions are reliable and have been well tested. This assumption is only included to justify our simulating classes of anomalies that are not representative of *programmer faults*. Also, if the versions are still suffering from reliability problems, then the versions are not yet ready for this analysis.
2. The voter is reliable. (We do not care which voting approach is implemented, as long as it is implemented correctly.)

2.2 Simulating a Faulty Specification for *n*-version Systems

Intuition suggests that wrong *n*-version specifications will always cause a *voter failure*. But when the voter does not change its vote after common specification errors are injected into the versions,

we must wonder why. Once we find those directives that if modified cause the voter to switch its vote, the question we want answered is "are we sure those directives are correct'?"

Specification errors are simply faulty specification directives that define internal computations. Simulating specification errors provides a prediction of how tolerant the voter will be to real specification errors concerning the *n*-versions.[3]

As an example, suppose that the specification has a directive to the programmers that says that the ALTITUDE variable is a function of f_1 + f_2. Suppose this function is wrong, and that it should be f_1 - f_2. Unless this if found before the code is programmed, this specification error will likely find its way into the versions. After all, the specification is usually the final authority on correctness.

Here, fault injection will be used to force a corrupt ALTITUDE value in each version on each test case. This is done in a manner reflective of a *common specification error*. This involves finding the appropriate source-code statements in the versions where the ALTITUDE computation directive is implemented, and then injecting the common anomaly into each implementation.

In fault injection, *perturbation functions* are the source code instrumentation utilities that inject *data state mutants* [8]. Data state mutants are corruptions to the values that particular variables have internally as the software executes. Developing new perturbation functions that simulate common specification errors in multiple versions was a key research task of this project.

For this example, a perturbation function will force the value of ALTITUDE to be reduced or increased by an equal amount, $\pm\Delta$, in all versions. If Δ were -30, then 30 would be subtracted from the value that each version computed for ALTITUDE = f_1 + f_2. It may well be that different versions have different values after executing the statements for this directive, and if so, we wish to retain that natural diversity. So for $\Delta = -30$, if one version had ALTITUDE equal to 40, then after fault injection, the value will be 10. If another version had ALTITUDE equal to 500, then it will get a value of 470.

After much thought, we decided that it is important to retain existing diversity in versions. We could have taken a different approach and selected a unique value *Q* (that no version had) and then given each version an ALTITUDE value of *Q*, But instead, we would do the following: ALTITUDE = f_1 + $f_2 \pm \Delta$.

The algorithm that we will employ for a single specification error to a numerical data type follows:

Algorithm for Simulating a Specification Error:

1. For some test case *i*, run the *n*-version system and store the output from each of the *n* versions in an array *A* of size *n*. Let *O* denote *V*'s decision based on the *n* version outputs in *A*.

2. Select a computation directive $\mathcal{C}$ from the specification *S* that is expected to be implemented in each version. (We will assume that $\mathcal{C}$ is implemented in each version, as we expect that each version is already highly reliable in isolation.)

3. For *i*, apply the standard perturbation function defined in [8] ($\pm$ 50% of current value as range for selecting new value)) to the result computed from the implementation of $\mathcal{C}$ in *one*

[3] One class that we cannot simulate easily here is incomplete specification errors, and thus in this paper, we will not address this class.

randomly selected version *e* from the *n*-version set.

4. In *e*, calculate the offset ($\pm\Delta$) between what the original result from $\mathcal{C}$ was and what the new value from $\mathcal{C}$ is after the internal value is perturbed. Note that Δ is a function of *i*, *e*, and the perturbation function.

5. For the other $n-1$ versions, the offset of $\pm\Delta$ is forced into the internal result computed by their implementation of $\mathcal{C}$.

6. Execute all *n* versions on *i* using the $\pm\Delta$-based perturbation function, and collect the output from the voter, *O'*, for this *i*.

7. If $O \neq O'$, then the voter was not tolerant to the specification error affecting $\mathcal{C}$. Also, if there exists *x* versions ($x \geq 2$) whose outputs equal *K* after fault injection but whose outputs in *A* did not equal *K*, then a single-input common-mode failure of severity *x* has occurred.

8. Perform the previous steps for a set of test cases and for each $\mathcal{C}$ keeping a count of the number of failures.

The possibility exists that some implementation of $\mathcal{C}$ is not executed when *i* is selected. If this occurs for all versions, then ignore *i* and select another input. Otherwise, perform the algorithm as explained, and perform the offset injection (in Step 5 of the algorithm) in those versions where $\mathcal{C}$ is exercised. As the number of versions increases that do exercise $\mathcal{C}$ for *i*, the likelihood that $O \neq O'$ decreases for this *i*. And if some versions are executing different calculations than their counterparts, then it is certainly possible that the voter will not produce the desired results for reasons other than identical programmer faults. (An example here would be specification ambiguities, which we will discuss in the next section.)

This algorithm simulates the situation of a precise "off-by-something" error ($\pm\Delta$) in directive $\mathcal{C}$ affecting the data states in each version at the appropriate place. By using an offset, and not just forcing a constant value into each version, we do not disturb other "natural" diversity that may already exist in the different versions. For a fixed $\mathcal{C}$, this algorithm will be applied for many different *i*'s and different Δ's. This suggests how sensitive voter *V* is to $\mathcal{C}$ in *S*.

Note that "off-by-something" errors are not the only specification errors that can be simulated. For example, we could have an "off-by-some-percentage" that simulates a multiplier effect. So instead of ($\pm\Delta$), we might want $\times K$, where *K* is a constant in (0, ∞].

2.3 Simulating an Ambiguous Specification for *n*-version Systems

In order to simulate an *ambiguous specification* directive, a directive that can be interpreted in several ways, we only need to make a small change to the previous algorithm.

Algorithm for Simulating a Specification Ambiguity:

1. For some test case *i*, run the *n*-version system and store the output from each of the *n* versions in an array *A* of size *n*. Let *O* denote *V*'s decision based on the *n* version outputs in *A*.

2. Select a directive $\mathcal{C}$ from the specification *S* that is expected to be implemented in each version.

3. For *i*, apply the default perturbation function defined in [8] to the result computed from the implementation of $\mathcal{C}$ in *one* randomly selected version *e* from the *n*-version set.

4. In *e*, calculate the offset ($\pm\Delta$) between what the original result from $\mathcal{C}$ was and what the new value from $\mathcal{C}$ is after the internal value is perturbed. Note that Δ is a function of *i*, *e*, and the perturbation function.

5. For some random number *r* (where *r* is in $[0..n-2]$), randomly select *r* versions, none of which can be *e*.

6. Take this set of *r* versions, find in each version where $\mathcal{C}$ is implemented, and force an offset equivalent to $\pm\Delta$ into the internal result from those implementations of $\mathcal{C}$.

7. Execute all *n* versions of the system (whether they had $\pm\Delta$ applied to their program states or not), and collect the output from the voter, *O'*, for this *i*.

8. If $O \neq O'$, then the voter was not tolerant to the specification ambiguity affecting $\mathcal{C}$ in the $r+1$ versions. Also, if there exists *x* versions ($x \geq 2$) whose outputs equal *K* after fault injection but whose outputs in *A* did not equal *K*, then a single-input common-mode failure of severity *x* has occurred.

9. Perform the previous steps for a set of test cases and for each $\mathcal{C}$ keeping a count of the number of failures.

This algorithm assumes that a computational directive $\mathcal{C}$ could be interpreted in one of *two* ways. This algorithm could be extended for 3 or more different interpretations of $\mathcal{C}$. This algorithm randomly selects which versions will be assigned the first interpretation and leaves the other versions alone.

3 Experiment

Our experimentation involved using five different versions of a controller for managing the traffic lights and turn arrows at a particular intersection. Admittedly, this result is for a toy example; we have tried to attain a real safety-critical *n*-version system that is written in C or C++ but had no success.

The original specification for this system was written by Adam Porter at the University of Maryland, and was modified and given to students at the College of William and Mary. The reason that modifications were made was so that manual correlation between specification directives and source code computations could be easily made. (Note that these versions were not intended to be part of a true *n*-version system, however team independence was mandated.)

In this specification, traffic can move going northbound (N), southbound (S), east bound (east to north (E), east to south (ES)), and north to west (NW). There is a traffic light controlling all northbound, south bound, and east bound lanes. There are also two turn arrows, one for the north to west turn lane and for the east to south lane. Figure 3 depicts the intersection.

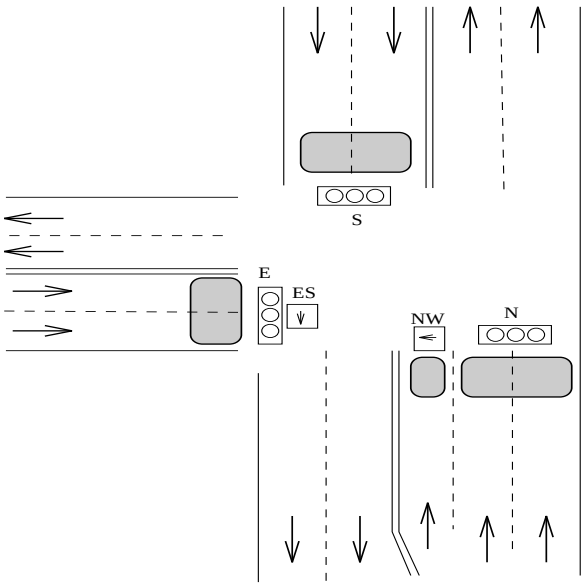


Figure 3: Intersection.

There are 4 sensors under the roadway: one for all eastbound lanes (E), one for all southbound (S) lanes, one for the northbound (N) lanes, and one for the north to west (NW) turning lane. A sensor emits an input signal only if at least one car is in the corresponding lane. The rate at which sensors emit signals is arbitrary. Each software version receives the sensor signals as input and then generates the appropriate traffic light controls (outputs) for all lanes at the intersection. The outputs indicate the color (GREEN, YELLOW, or RED) of every traffic light to reflect the current traffic flow.

This experiment employed the algorithm for simulating a specification error. In the specification, there were 11 key calculations that the programmers were instructed to handle inside the traffic light software:

1. **North Green** Give traffic headed North the green light.

2. **North Yellow** Give traffic headed North the yellow light.

3. **North Red** Give traffic headed North the red light.

4. **East Green** Give traffic headed East the green light.

5. **East Yellow** Give traffic headed East the yellow light.

6. **East Red** Give traffic headed East the red light.

7. **South Green** Give traffic headed South the green light.

8. **South Yellow** Give traffic headed South the yellow light.

9. **South Red** Give traffic headed South the red light.

10. **Northwest Arrow** Give traffic heading North the turn arrow to go West.

11. Southeast Arrow Give traffic heading East the turn arrow to go South.

In our experiment, we executed twenty test suites on the five version system. Each test suite is a chain of events at the intersection, where an event is either a single sensor emitting or multiple sensors emitting simultaneously. Since there were 11 $\mathcal{C}$s for which specification errors could be simulated and a total of 510 inputs from all of the test suites, the *n*-version system was executed for a total of 5610 test cases. For each of the 5610 test cases, we collected the results generated from the five versions without any perturbations to verify that there was a majority ruling on the output of the system.

If there was not a majority when the voter compared the outputs from each of the versions, then the test case was never executed with perturbed specification directives. The test case was omitted simply because the versions were not able to agree on the output under "normal" circumstances. In most existing *n*-version systems, the individual versions are thoroughly tested before integration into the *n*-version system, and thus the likelihood of undetected programmer faults is slim. The versions used in this experiment were not tested as thoroughly as in real safety-critical *n*-version systems, however, there was a majority agreement from the five versions for 69% of the test cases.

If there was a majority output from the versions, then we executed the five version system for every specification directive that was selected to be perturbed. Of these test cases, 23% resulted in the voter making a different decision. The results from the analysis of applying the algorithm for simulating a specification error are summarized in Figure 4: the indices along the x-axis are the eleven specification directives and the y-axis corresponds to the number of single-input common-mode failures that occurred. Note that the specification directive that caused the voter the most problems was the directive handling the red light for Eastbound traffic; it caused the voter to fail 164 times.

The reason for this appears to be complexity in the specification. There were test scenarios where events at traffic sensors occurred simultaneously. For instance, two cars might request a Northwest turn and a Southeast turn from the controller at the same instant. These more complicated input events confused several of the programming teams, and it appears to have made the East Red directive extremely sensitive to errors. Also, the test cases that were employed exercised the East Red directive more frequently. Hence if you have access to an operational profile, that information should be used for even more accurate predictions concerning directive sensitivities.

Had this been a real system and had simultaneous events like this occurred frequently during daily operations, then this analysis would have isolated the specification directives that were more capable of making the traffic intersection unsafe. This suggests which specification directives need greater attention during validation.

4 Summary

This paper has looked at a method for predicting how likely it is that an *n*-version software system will trip up the voter when the specification is defective. By knowing the sensitivity of voters to different specification anomalies, we can isolate those parts of the specification must be "solidly" correct. If those portions of the specification that need to be correct are not, our approach predicts that common-mode failures are more likely to occur.

We have focused on systems that employ design diversity. There is no reason that fault injection cannot be used to simulate specification anomalies for single-version systems.

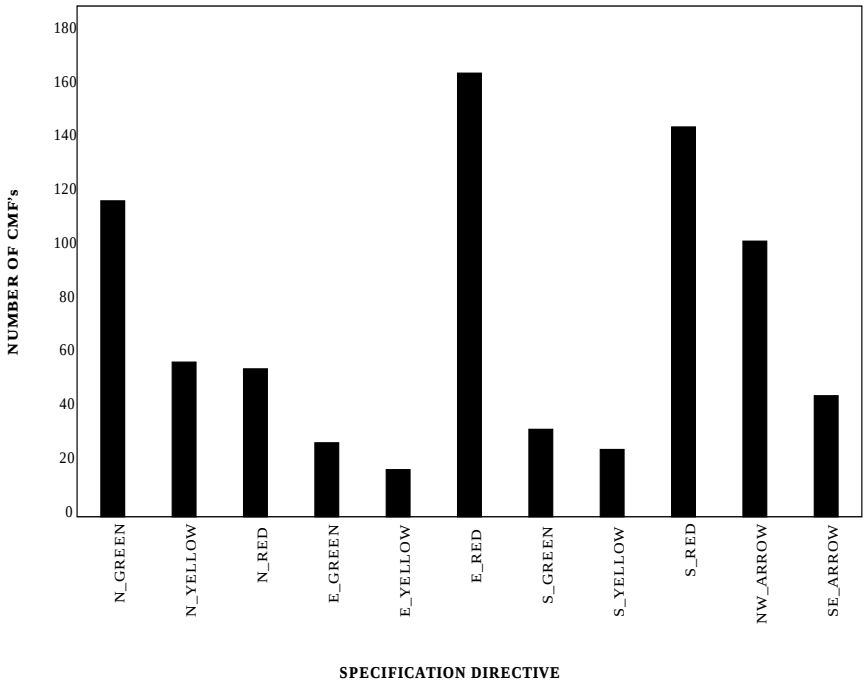


Figure 4: Specification directives and single input common-mode failures

It is known that earlier elimination of errors in the life-cycle is cost-prudent. Little research has ever been done to our knowledge that targeted eliminating only those specification problems that will have a detrimental impact on the system. The approach we have outlined here is a first step in that direction.

We admit that it would be preferable to have the results of this analysis much earlier in the life-cycle. But at this point, we do not know how to achieve that.

Acknowledgements

This work has been partially supported by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160. The authors thank Prof. Adam Porter (University of Maryland) for supplying us with the original specification used in the experimentation. Prof. Porter also provided Figure 3 to us. The authors thank Frank Charron for his help in building the tool that was needed for the experimentation.

References

[1] D. Briere and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls - a family of fault-tolerant systems. In *FTCS 23*, pages 616–623, Toulouse, June 1993.

[2] S. Brilliant, J. Knight, and N.G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Transactions on Software Engineering*, SE-16(2), February 1990.

[3] Federal Aviation Authority. Software Considerations in Airborne Systems and Equipment Certification, 1992. Document No. RTCA/DO-178B, RTCA, Inc.

[4] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley, 1989.

[5] Peter Ladkin. A320 Flight-Control Computer Anomalies *Software Engineering Notes Risk Forum* 18(78).

[6] J. Knight and N.G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[7] P. Traverse. Dependability of digital computers on board airplanes. In *DCCA 1*, Santa Barbara, CA, August 1989.

[8] J. Voas. PIE: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Eng.*, 18(8):717–727, August 1992.

[9] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, July 1997.

[10] J. Voas, A. Ghosh, F. Charron, and L. Kassab. Reducing Uncertainty about Common-mode failures. To appear in *Proceedings of the International Symposium on Software Reliability Engineering*, Albuquerque, NM, November, 1997.

Making Implicit Requirements Explicit, An Application of the Peeking Methodology

Leslie Allen Little - Aztek Engineering
2477 55th St. Suite 202, Boulder CO 80301
lal@aztek-eng.com www.aztek-eng.com

1. Abstract

An eternal problem with the software universe is the continual addition of *implicit* requirements to the software product. Implicit requirements are defined as product requirements that are not explicitly defined, yet exist. Ideally, any commercially available product should not contain implicit requirements. Practically speaking, all products do.

Peeking is a methodological process developed at Aztek Engineering that is embodied in a set of tools and processes. It was originally developed in an attempt to alleviate the difficulties associated with two significant problems that plague the vast majority of all software products—namely the inability to

- define all product capabilities accurately as explicit requirements
- reduce system testing efforts without reducing product quality

The concept behind peeking is simple—build a database of product requirements linked to the source code that embodies those requirements. Application of this method on two projects to date has resulted in the discovery of 25% and 71% more requirements than were originally specified.

2. Keywords

Keywords/Phrases: Quality Assurance, System Test, Test Methods, Traceability, RDBMS, Requirements, Black Box Testing, Requirements Methodology

3. Biographical Sketch

Leslie Allen Little is the senior Quality Assurance Engineer at Aztek Engineering. He manages a quality assurance group whose primary responsibility is system testing for a large PBX. His primary areas of expertise are Quality Assurance, System Testing, Telecommunications, Software Development and RDBMS.

Mr. Little holds an AAS in Computer Science, a double major (BSBA) in Philosophy and Computer Information Systems from the University of Southern Colorado and a MS in Telecommunications from the University of Colorado at Boulder. He has previously published articles in the System Testing, Quality Assurance and Requirements arenas.

Mr. Little has worked in the Software Development/Quality Assurance areas for 12 years; 8 years as Member of Technical Staff with AT&T Bell Laboratories (now Lucent), and 4 years as a software Quality Assurance Engineer with Aztek Engineering

Table of Contents

Table of Tables

Table of Figures

4. Introduction

The quality of a software product can be measured in terms of the

1. Completeness of the product definition
2. Percentage of defects that reside in the product be they known or unknown

There are numerous quality metrics presented in literature concerning the tracking of discovered defects and the estimation of remaining defects. Each of these techniques assume that an understanding of how the product should perform is somehow known a priori. It should be recognized that one cannot begin to measure the percentage of problems, be they known or unknown, without first defining the product. Recognize that we are not talking about a static product definition, but instead a continually evolving product definition. The inability to maintain a documented understanding of product capabilities, as those capabilities continue to change, makes this the Achilles heel of the software industry.

In an attempt to more fully define the software product and to increase the ability to rationally test the software product, the Peeking Methodology was born[1]. This methodology provides a simple means for capturing product capabilities as they are continually defined and redefined throughout the product life cycle. Making it reasonably simple increases the likelihood of it actually being implemented and sustained throughout a product's life cycle.

This paper seeks to examine and explain the methodological process of peeking.

5. What is the Peeking Methodology?

The Peeking Methodology is the placement of a methodological process upon software's embodiment of product requirements. It is accomplished by defining then examining all source code segments, and then tying those code segments to product requirements. Code segments are defined as consecutively grouped lines of code having a similar or same purpose. A single code segment usually encompasses an entire function and sometimes even an entire source code file. Some functions, however, consists of multiple code segments. Similarly, header files often have distinct parts that also may be considered to be different code segments. Product requirements are defined in a hierarchical parent/child relationship ranging from high leveled requirements to detailed ones.

When the Peeking Methodology is applied to a large project, recognize that this can be a massive undertaking. This can be alleviated somewhat by instituting a policy limiting the scope of the peeking process.

The peeking process easily fits into the vast majority of existing software development models. The methodology is geared towards the collection and identification of relations between requirements and code segments. The act of performing this linkage results in the reduction and eventual elimination of implicit requirements.

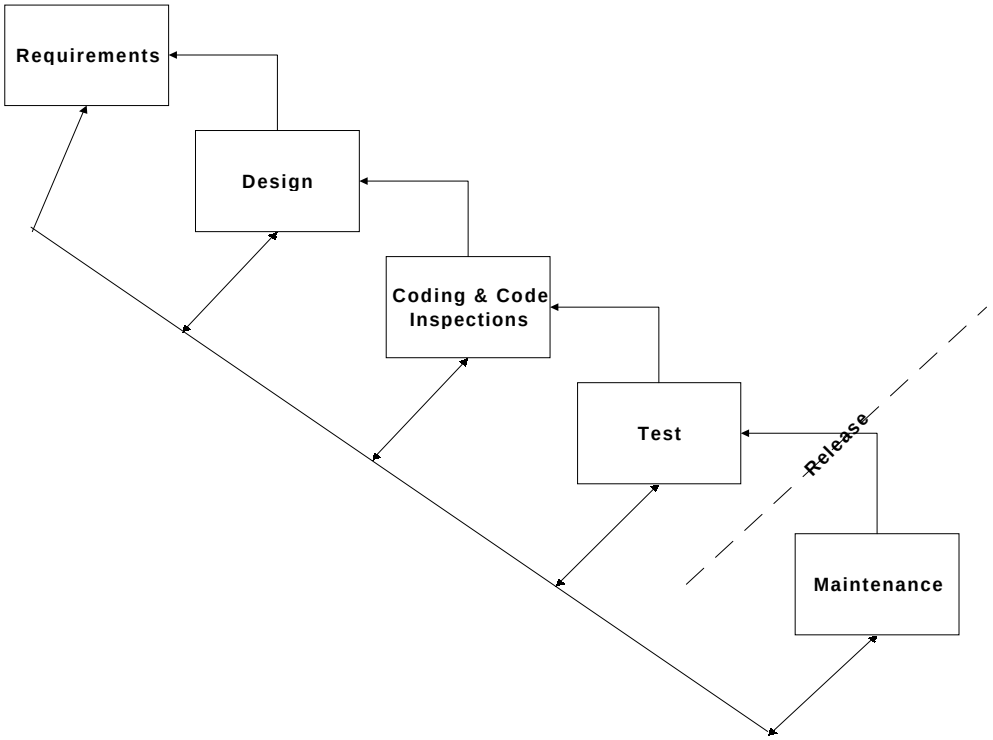
[1] The term peeking was coined as a result of the author's early work on this methodology. That work attempted to find a way to more easily determining which parts of an existing product are at greater risk of breakage by the delivery of new functionality and capabilities. The peeking methodology was initially established to aid in this discovery. As such, the term *peeking* was aptly and affectionately applied to the methodology because it provided the system test engineers a simple way to peek inside the black box normally associated with system test methodologies.

6. Where Peeking Fits into the SW Development Model

In theory, all software is created for the sole purpose of implementing some requirement or set of requirements, thus requirements are normally the beginning point for all software development models. Traditionally, the spiral or the classical waterfall models are normally cited as the life cycle process models in literature. Variations on these model leads to the *Incremental Delivery Approach* and the *Evolutionary Development Approach*[I] models, both of which extend the *Waterfall* model to add planned future releases into the model rather than to assume all work is done initially[2]. Typically, any development of any serious size, inevitably uses some variation of the *Incremental Delivery* or the *Evolutionary Development* approaches as their life cycle models.

At Aztek, we utilize a modified evolutionary delivery model as presented in Figure 1. This model has feedback loops inserted for each phase of the product life cycle.

Figure 1 - Aztek Software Development Cycle



Referring to Figure 1, the implementation of the Peeking Methodology is concentrated in the *Coding and Code Inspections* portion of the life cycle model. This is where the peeking data collection effort occurs.

This does not mean that data collection doesn't occur during other phases, however. Given the feedback loops that exists throughout the produce life cycle, peeking data collection occurs during every portion

[2] The Incremental Delivery Approach expects incremental deliveries of capabilities over time and plans for it by altering the process model to repeat itself starting over from the detailed design level. Similarly, the Evolutionary Development Approach expects new capabilities over time yet it repeats the entire life cycle model over and over starting over with the user requirements level.

of the product life cycle—any time software changes, it comes under the scrutiny of the Peeking process.

Initially, as shown in the diagram, the Peeking Method depends upon requirements having been created and recorded in the *Requirements* and *Design* phases. To facilitate the Peeking Method, it is assumed that a requirements management process exists along with a set of requirements management tools. These tools facilitate requirements creation and collection throughout the product life cycle. Application of the Peeking Methodology enforces requirements conformity and discovery while simultaneously collecting the data that results from this process for later usage.

7. How and When Is Peeking Accomplished?

One can establish the Peeking Methodology at the beginning of a product, or adopt this process once a product has already begun. Either method works, although, it is simpler and offers more benefits to implement the Peeking Methodology as early as possible.

In either case, their are some prerequisites to effectively implementing the Peeking Methodology. Without these prerequisites, the benefits of peeking are mitigated. These prerequisites include:

1. A software methodology that considers and incorporates a change management system for requirements.
2. A requirements management system that is implemented as a database.
3. Adequate tools support.[3] There are two sets of tools required:
    - Tools to support development engineers in the process of identifying requirements to source code segment mappings. This includes a simple way to match requirements with source code segments; a call tree diagram of files, functions and functional dependencies, and a method of identifying and tracing call and data flows.
    - Tools to support the identification of changed source code segments and the requirements affected during a particular product life cycle phase. These tool are needed when the Peeking Methodology is used to narrow regression test strategies.

To properly implement the Peeking Method, the following steps are required.

1. Put the tools in place that aid in collecting peeking data from source code[4]
2. Establish the requirements database[5] and process controls to keep the database up-to-date.
3. Establish the peeking database and process controls to keep the database up-to-date.
4. Follow through with the processes established.

[3] The major piece missing in a widespread implementation of this methodology is the creation of useful tools support for the process. The author recognizes these deficiencies and is working towards a remedy to this situation.

[4] As mentioned previously, the tools necessary to implement peeking on a wide spread basis are still under development at Aztek. The application of the peeking method to date has been performed with ad-hoc tools. No time table has been set as to the completion of these tools.

[5] The author has completed development of a set of tools that implements a requirements database management system. This RDBMS system, called ReqTrack, is available (currently free) from the author. Please address requests to lal@aztek-eng.com or via fax at 303-786-9190.

8. Peeking Implementation Details

Although the process ideas are the same, the Peeking Methodology implementation differs slightly depending upon the development language and operating environment utilized. To date, two projects have had this methodology applied to them. The first was a multitasking, real time, embedded systems application written in C; the second was a multi-user graphical user-interface (GUI) application written in Visual Basic™ that utilized a Microsoft Access™ database, objects, and widgets. The author sees no reason to doubt that all other languages and environments can also have this methodology applied to them with minor implementation modifications.

The following sections explain the peeking database tables and the general process and implementation details. As the authors experience level grows in applying this methodology to more and more differing environments, more general implementation details may be discovered just as tweaks to the existing processes may be necessary.

8.1 Peeking Database Tables

The peeking database is implemented as a relational database that works in conjunction with a requirements database. In fact, in the implementations performed by the author, the peeking tables are simply additional tables to ReqTrack, the existing requirements management database system. The peeking tables consist of three tables as shown in Table 1.

Table 1 - Peeking Tables in RTC

Table Name	Field 1	Field 2	Field 3	Field 4
Source-Files	Code Segment	Source Code File	Modification Date	
Code-To-Requirements	Code Segment	Source Code File	Requirement Number	
Parent-Child	Calling Code Segment	Calling Source Code File	Called Code Segment	Called Source Code File

The table and field names are relatively self explanatory. *Source-Files* embodies the relationship between code segments and source code files. The modification date field is not historical, but simply contains the last time that the code segment has changed. The key value for this table is the first two fields.

Code-To-Requirements contains the mapping between code segments and requirements via a unique requirement number. The key value for this table is all three fields.

The last table, *Parent-Child*, embodies the relationships that exist between parent code segments and child code segments.

8.2 Source Code Additions, Deletions, and Modifications

Peeking attempts to control all changes made to source code and to record the impact of those changes to requirements regardless of whether the source code is being added, deleted, or changed. In the beginning, there are only additions of source code. Later comes modifications and deletions of existing

source code. In each case, peeking is applied at the same time that code reading, walk-through or code inspections occur.[6]

8.2.1 Source Code Additions

The peeking process, as applied to source code additions results in

1. Uniquely identifying and grouping all source code being delivered[7] into code segments. For applications that follow structured approaches, the guideline is simple—each function and/or subroutine is a code segment[8].
2. Examining each line of code and determining the requirement(s) that are embodied in the code segment, noting these relationships in the appropriate peeking database table as explained later. Examples of this process are presented in section 8.3

8.2.2 Source Code Deletions

When source code is deleted, the peeking database must be consulted for references to the deleted code segments. This is relatively easy since the *Source-Files* table in the peeking database contains the mapping between source code files and source code segments.[9]

8.2.3 Source Code Modifications

In the case of modifications to source code, the process essentially requires the user to repeat the process of examining each line of code to determine the impact. The impact of the changed source code may or may not have an impact on the peeking database depending upon the modifications that occurred.

8.2.4 Structuring Source Files to Support Peeking

In general, it isn't necessary nor productive to force developers into some new method of writing their code to support peeking. In the early applications of this process, the need to restructure code to support the process has been unnecessary. As the author develops tools that seek to automate the data collection process, such a restructuring might prove necessary. Ideally, this should be avoided. It will only be as a last resort that such measures will be taken.

8.3 Rules and Examples - Applying the Peeking Methodology

The peeking process is not an absolute process just as specifying requirements contains few absolutes. The degree of granularity to which the process is applied must be reflective of the degree of control and granularity desired by those implementing the process. Therefore, to aid in the practical application of the peeking process, guidelines are provided about how to apply the process—not about to what degree the process should be applied. Too much detail and documentation can kill a project! Likewise, too little detail and documentation can have the same effect. The author leaves it to the reader to determine that elusive optimal level of detail required.

---

[6] Since the peeking methodology has yet to be implemented on a widescale basis, the specifics of how to enforce the necessary changes have not been addressed. The author expects that whatever the process followed, it will most likely depend upon the flexibility of the institution implementing it.

[7] There are certain exceptions as indicated later in this article. See section 8.3.

[8] If the application is multithreaded and one intends to applying peeking to the multithreaded code also, then the main loop function that handles messaging stimuli needs to treat each messaging case as a code segment also.

[9] One small caveat is that of source code segments moving from file to file. This also affects the peeking database. Tools support for both of the above conditions need to be developed to automate these tasks.

The following paragraphs espouse the rules associated with the application of the Peeking Methodology along with code segment samples that seek to crystallize the process.

8.3.1 Header Files

In the C language, header files are used as common repositories to

1. Declare structure definitions
2. Make prototype declarations
3. Define constants
4. Define macros
5. Define and populate arrays that sometimes serve as lookup tables for state machines

With respect to the Peeking Methodology, the first three of these offer little or no value to peeking. The latter two, however, are quite different.

8.3.1.1 Macro Definitions

Macro definitions should be treated just the same as function calls—i.e., as code segments. They essentially are code. An example of this might be the following macro definition taken from a header file called SWI.h

```
/*
 * SWI_CONVERT_BDFRAM_CHANNEL:
 * macro to get a swi or r2 channel id from co board id & framer id
 * a simple algorithm: 2 E1s per board, 2 boards per COT
 * (divide channel by (2 * channels per e1 = 64)) + ef0 board id
 * (((brdId - EF_0_BRDID)<<6) + ((framerId - CO_FRAMER_0_ID)<<5))
 */
#define SWI_CONVERT_BDFRAM_CHANNEL( brdId, framerId ) ((CHANNEL_ID)     \
      ((((unsigned int)brdId - (unsigned int)HW_EF_0_BOARD_ID) << 6) + \
      (((unsigned int)framerId - (unsigned int)EF_CO_E1_0) << 5)))
```

The corresponding entries in the appropriate peeking tables would look like:

	Code Segment Name	File Name	Requirement #
Code-To-Requirements	SWI_CONVERT_BDFRAM_CHANNEL	SWI.h[10]	S348

8.3.1.2 Lookup Tables

Lookup tables definitions that have been initialized as part of the declaration also cannot be ignored. Without these definitions, application code utilizing the arrays cannot operate correctly. The array

[10] For simplification, full paths to the code segments are not given in the examples. The author has found it useful to do so in actual implementations of the process though.

definitions must be tagged by their array names and tied to the appropriate requirement in the peeking tables. The following coding example exemplifies such a case:

```
R2_ST_FUNC_T    *r2trkTbl[MAX_TRK_STATES][MAX_TRK_STIMS] = {
      /*************************************/
      /* State: 0 IDLE                     */
      /* CAS BITS:    OUT: 10      IN: 10  */
      /* Active Timers:      none          */
      /*                                   */
      /* Channel is idle in all directions */
      /*************************************/
   /*   CAS00        CAS01        CAS10  */
       {r2Idle00,     r2Idle01,      r2Null,
   /*   CAS11        RECOVER    GET_CPC- */
         r2Idle11,  r2RecoverErr,   r2swiErr,        …. More and more of this …
```

The corresponding entries in the appropriate peeking tables appear as follows:

	Code Segment Name	File Name	Requirement #
Code-To-Requirements	R2_ST_FUNC_T	r2States.h	S102
Code-To-Requirements	R2_ST_FUNC_T	r2States.h	S221
Code-To-Requirements	R2_ST_FUNC_T	r2States.h	S382

Note: There are three entries for this code segment. The reason for this would be to express that this code segment is the embodiment of three separate requirements.

8.3.2 Library and Function Files

The following rules are applied to all other source code files including functions and libraries:

- All data and function definitions contained in code segment are ignored.
- All structure definitions contained in code segments are ignored **unless** they are used as lookup tables in which case they should be handled as explained in section 8.3.1.2.
- General statements and flow control are treated as single code segments and examined as to the requirements that they embody. In the following example, comments by the author are inserted using the arrow symbol (⇒) in the left hand margin along with a different font size. For context, the function *formatCotIpMsg* is used to format messages that will be sent along an communications path.

```
DU_MSG *formatCotIpMsg( STD_IP_MSG ** hIpMsg, UINT16 len)
```

{

⇒ This is the function definition which will be used to identify the code segment.

```
DU_MSG    *     pCotMsg;
STD_IP_MSG *    pIpMsg;
pIpMsg = (STD_IP_MSG *) OsGetMem(len);
```

⇒ It isn't necessary to do anything about pointer definitions as indicated by the first two lines above. The third line, however is interesting. It represents a call to some low-level function that apparently allocates memory to this function of length *len*. Whether or not this should be tracked through the peeking process depends upon the level of detail that has been decided upon for the project implementing this process. This might be determined to be of a level that is too low and of little importance. If so, it would be ignored. If not, then it would be tracked and one would expect to find some requirement in the requirements database that specified the memory allocation and de-allocation from some common memory pool. Let us assume that it is being tracked. The entries into the peeking tables would appears as follows:

	Calling Code Segment	Calling Source Code File	Called Code Segment	Called Source Code File
Parent-Child	formatCotIpMsg	mntc.c	OsGetMem	util.c

	Code Segment Name	File Name	Requirement #
Code-To-Requirements[11]	OsGetMem	util.c	S322

```
 if (pIpMsg != NULL)
{
    pIpMsg->type = COTI_SEND_SLC_MSG;
    pIpMsg->status = 0;
    pIpMsg->len = len - sizeof(STD_IP_MSG);
    pIpMsg->dest = COT_ADDRESS;
    pIpMsg->src = DU_ADDRESS;
    pIpMsg->pMsg = (UINT8 *) pIpMsg + sizeof(STD_IP_MSG);
    pCotMsg = (DU_MSG *)pIpMsg->pMsg;
    pCotMsg->q.slc.rt = DU_ADDRESS;
    pCotMsg->q.slc.port = DU_MNTC_PORT;
    *hIpMsg = pIpMsg;
    return pCotMsg;
}
```

[11] Note that no entry is made for the *formatCotIpMsg* function at this level. That mapping will take place by virtue of the *Parent-Child* table mappings if and when it is needed. Requirement S322 would be the low-level requirement that requires that memory allocations are present and properly handle requests.

```
        else
        {
            ReportErr(DU_ERR_BUF_GET_FAILURE, _LINE__,__FILE__);
            *hIpMsg = NULL;
            return NULL;
        }
    }
```

The above conditional expression is most likely of interest to the those applying the Peeking Methodology since it indicates that a decision is being made. In the above case, if the message that the pointer points to is not null, then data is moved into the memory that the pointer points to and the pointer is returned to the calling function. If not, then an error is reported and a null pointer is returned. This code segment actually has the following entries associated with it in peeking tables:

	Calling Code Segment	Calling Source Code File	Called Code Segment	Called Source Code File
Parent-Child	formatCotIpMsg	mntc.c	ReportErr	errlib.c
Parent-Child	formatCotIpMsg	mntc.c	OsGetMem	util.c
Parent-Child	mntc	mntc.c	formatCotIpMsg	mntc.c

	Code Segment Name	File Name	Requirement #
Code-To-Requirements[12]	formatCotIpMsg	mntc.c	S632
Code-To-Requirements	mntc	mntc.c	DUHW03
Code-To-Requirements	ReportErr	errlib.c	DUSW05
Code-To-Requirements	ReportErr	errlib.c	DUSW10.1
Code-To-Requirements	ReportErr	errlib.c	I-111
Code-To-Requirements	ReportErr	errlib.c	I-151
Code-To-Requirements	OsGetMem	util.c	S322

The important points from the above discussion are these

- Lower-level code segments, when changed, have a potentially enormous affect on the stability of the system. Since lower-level code segments are at the root of the tree, all up stream code are quite dependent upon the lower-level code segments for the successful fulfillment of its requirements.
- Some code segments may not be tracked in the peeking tables. In the above example, the *OsGetMem* code segment may or may not actually be tracked. Whether it is tracked is associated

[12] The description for requirement number S632 would require that a specific error messages be generated as a result of resource exhaustion. It could very well be that many other function also are linked to S632. There are no prohibitions to tagging multiple code segments with the same requirement. When software is modularized, for re-use purposes, this often happens.

with the decision by those implementing the process about what level of requirements detail is enough. At some point, too much detail is being captured to be of use and must be left out. The key to the Peeking Method and to requirements definition in general is to appropriately determine where that boundary is.

An important point not illustrated but worth mentioning is that not all code segments will have requirements directly tied to them. In the above example, if the *formatCotIpMsg* code segment illustrated above had no conditional expression, then there would be no requirement directly tagged to the *formatCotIpMsg* code segment. If you follow the parent/child relationships for this code segment, however, you will notice that it indirectly affects the successful or unsuccessful fulfillment of 5 other requirements.

9. Peeking Methodology Mechanics

In every methodology, various processes and mechanics are introduced to aid in the successful implementation of the methodology. Peeking is no exception to this. The following sections discuss some of these process issues.

9.1 Ongoing Feature Development and Bug Fixes

In the maintenance phase of a product life cycle, normally feature development continues along with corrections for discovered faults. Peeking is not adversely affected by this phenomenon, in fact it thrives. Given that one of the prerequisites to Peeking is to have a requirements database and requirements management process in place, all that is required is that the maintenance code be inspected and the peeking database updated.

9.2 Complexity and Source Code Segments

Some source code segments are more complex than others. Complexity is not an issue for peeking. Regardless of the complexity of source code segments, requirements continue to apply to the code segment. If one cannot understand what requirements apply to the code, then the code needs to be rewritten.

An important rule to use when creating the *Code-To-Requirements* peeking table is that too much detail can defeat the entire method. The examples given throughout this paper have attempted to show, among other things, how to deal with the complexity issues that may arise.

One has to make a judgment call on how much detail is enough. When analyzing source code, err on the side of too little detail rather than too much. This should be the general rule of thumb.

9.3 Variations in Source Code Languages

To date, peeking has been applied to two widely divergent source applications. Very little modification to peeking procedures was needed. The author feels that the Peeking Methodology is source code language independent. After all, all programming languages seek to effect the same thing, the embodiment of some set of requirements.

9.4 Uniting Product Capabilities with the Source Code

Getting the mapping between requirements and source code correct is critical to the validity of the Peeking Method. There are many cases where one code segment can be used to embody many requirements just as there can be many code segments that work in concert to fulfill one particular

requirement. One cannot overlook the importance of the initial data collection or continued data collection. Peeking has its costs just as all other methods. It requires a long term commitment and a proper allocation of time and effort. As we will see later, the actual cost of applying the method is relatively cheap in overall product costs.

9.5 Maintaining Product Capability and Code Segment Linkages

It must be recognized that the product being tested is a moving target; requirements evolve, are added, deleted, etc. Maintaining an accurate link between product capabilities and source code segments is difficult, yet critical. Ideally, code segment changes are examined during code inspections and again at the time of source code delivery to the common repository. Tools to assist in this process and other processes associated with this methodology are being developed.

10. Applying the Peeking Method in the Real World

The following sections examine the practical application of the Peeking Methodology performed by the author. The first amounted to an early case study which was conducted in late 1996 and early 1997 on a large existing application called the Distribution Unit (DU). The initial purpose of the study was to determine if and how the Peeking Methodology could reduce regression testing in the test phase of the software product life cycle. The results were inconclusive but the act of applying the methodology pointed out the requirements discovery and documentation value of applying the methodology.

More recently, the Peeking Methodology has been applied to a small, GUI based internal tools project called AzTrack. The application of the methodology started from the beginning of the project and continues to be maintained.

The results of each of the above applications of the Peeking Methodology follow.

10.1 The Distribution Unit (DU)

The DU was selected from a choice of three real time, multiprocessing, embedded system components that work in concert via a communications link to perform Private Branch Exchange (PBX) capabilities. The application software was written primarily in *C*™ with a minor amount of *Assembler*™ code.

The code base consisted of 19,953 lines of code, of which 9,977 lines are Non-Commented Source Code Lines (NCSLs). It had reasonably limited capabilities that were originally defined in 58 requirements. These 58 requirements ranged from high-level to reasonably detailed ones.

10.1.1 Objectives of Peeking for the DU Study

The DU component was elected for the peeking study because of its simplicity, relative code stability, and the fact that there was an easily identifiable period that could be examine while the product was in the maintenance phase of its life cycle. The study was to answer the following questions:

1. When system testing the product, are the number of tests requiring execution reduced as a result of peeking?
2. Is software quality compromised as a result of applying the Peeking Methodology?
3. Is the product better defined as a result of applying the Peeking Methodology?

Results of this study for the first two questions, if the reader is interested, can be examined in an article presented at the Proceedings of the Sixth International Conference On Software Testing Analysis &

Review.[II] In summary, the results appear to support the notion that test case re-execution would be reduced without compromising quality but were inconclusive mostly due to the scarcity of good data on subsequent product defects.

Results for the third question, "Is the product better defined as a result of applying the Peeking Methodology" are presented below.

10.1.2 Results of Peeking on Product Quality

The results of retroactively applying the Peeking Methodology to the DU software easily confirmed that product quality is enhanced. Glaring examples of missing requirements, such as how the LEDs behave on the boards controlled by the software were discovered. As a result of the study, a number of existing requirements were discovered to have never been implemented while others required wording modifications—modifications that affected the meaning of the requirements. Table 2 summarizes the raw numbers and percentages represented by these findings.

Table 2 - Requirement Modifications Needed Due to Peeking Study On DU

New Requirements	Requirements Never Implemented	Requirements Needing Wording/Content Modifications
15 (25%)	4 (7%)	3 (5%)

10.2 AzTrack

AzTrack is an internally developed defect tracking tool used to record and track software and hardware problem reports. The tool is a GUI application using Microsoft Access as the database engine and GUI builder. Application code was developed in conjunction with the controls and attributes automatically provided via Access. The application code is written in Visual Basic. The product was reasonably well defined and a summer intern from the University of Colorado was recruited to perform the work during the summer of 1997.

The normal Aztek version of the waterfall life cycle process model was followed with a requirements document being produced and accepted, a prototype product produced, and a detailed design document completed. The requirements that resulted from these phases were documented and placed into a requirements database. During the code inspection phase, the additional Peeking Methodology step was included and statistics kept with respect to this effort. The product is now in a production setting, and is about to undergo its first minor release.

The following paragraphs discuss the application of the Peeking Methodology on this project.

10.2.1 Objectives of Peeking for the AzTrack Study

The objectives of applying the Peeking Methodology to AzTrack product were to quantify the following:

- What are the incremental labor costs associated with applying the Peeking Methodology
- The quality of the product via increased requirements/product definition
- If defect removal rates are directly associated with the application of the Peeking Methodology
- If decreased regression testing costs could be expected as a result of applying the Peeking Methodology

- If the Peeking Methodology could be applied to a dramatically different application development environment

10.2.2 Overview of Results from AzTrack Study

The process followed by the summer student, as outlined above, was to create the requirements with the help of the author, design a product to meet the requirements, and then to implement the product. Table 3 contains statistics collected from the application of the Peeking Methodology to the AzTrack project. Examining these statistics in light of the objectives of the study reveal the following:

1. Assuming that requirements are specified for a project, are uniquely identified, and are readily available, the incremental cost of applying the Peeking Methodology is reasonably nominal when considering the benefits gained.[13]

2. As a result of applying the peeking process, 14 new requirements were identified and documented. This excludes any added requirements not clearly identified through the peeking process and represents 71% more requirements. Clearly the process results in a more well defined product.

3. Defects were uncovered in the product that was tested as a result of being identified by applying the Peeking Methodology. One cannot make claims concerning the likelihood of these problems being found regardless of whether the process was utilized or not. One can, however, contend that the risk of these defects being discovered in a production setting are greater otherwise.

4. At this point in the study, there is no data available to either support or refute the notion of decreased regression testing costs. As the product moves through the maintenance phase of its life cycle, the defect removal efficiency costs can be compared with other like products in an attempt to answer this question.

5. The application of this methodology to the radically different application development environment was not overly difficult. The methodology continued to provide dividends as discussed above, and only minor modifications were needed to reach these goals.

[13] Note that these numbers may be dependent on project size since one could assume that the larger the project, the more difficult the process. Also note that the incremental effort required for peeking could easily decrease when tools are implemented that aid in the discovery process.

Table 3 - Results from Aztrack Peeking Study

Description of Data Collected	Raw Numbers
Total time spent inspecting the application code	14 hours[14]
Total time spent applying the Peeking Method to the application code	2 hours[15]
Number of initial requirements	21 requirements
Number of added requirements not specifically resulting from the application of the Peeking Methodology[16]	4 requirements
Number of added requirements resulting from the application of the Peeking Methodology	14 requirements
Number of modified requirements resulting from the application of the Peeking Methodology	0 requirements
Number of deleted requirements resulting from the application of the Peeking Methodology	2 requirements
Number of application defects found during system testing the product for only original requirements	0 defects
Number of application defects found during system testing the product for all added, modified and/or deleted requirements resulting from peeking	2 defects
Total number of defects found in the application once in production	1 defect
Total lines of code in application	987 LOC

10.2.3 Added Requirements for AzTrack as a Result of Peeking Methodology

As shown in Table 3, 14 requirements were added and 2 deleted as a result of applying the Peeking Methodology. Rather than bore the reader with a recitation of the requirements, suffice it to say that 8 were high-level requirements while 6 were low-level requirements.

One may ask, is this typically what happens in a development environment? The author would suggest that the answer is an unequivocal **yes**, it really is what happens. During the implementation process thoughts are thrown around between developers about things that might be better than was originally designed; developers learn new skills and new features that they can easily add to the product that, make it *better*, etc.

The most common requirement defect uncovered using the Peeking Methodology is that of the *missing requirement*. In both applications of the methodology this proved to be true.

[14] Fourteen hours may seem excessive for inspecting approximately 1000 LOCs in terms of industry averages, but the author had to familiarize himself with the language as part of the exercise.

[15] This represents the time it took to search for a requirement and to make the needed notations in the peeking database tables.

[16] These essentially were added features that were not originally considered in the requirements and design phases.

10.2.4 Aztrack, Another Code Segment Example

The following is one of several code segments comprising the AzTrack application code. Comments by the author are inserted using the arrow symbol ⇒ in the left hand margin along with a different font size.

```
Option Compare Database
Option Explicit
Dim Commit_Flag As Boolean
```

⇒The above statements are simply declarations global to the file containing the following subroutine, *EventHistoryButton_Click*. Peeking does not attempt to tie local or global declarations with requirements.

```
Sub EventHistoryButton_Click()
```

⇒This is simply the subroutine definition. No actions taken for this declaration.

```
On Error GoTo Err_EventHistoryButton_Click
```

⇒*On Error* is a general directive to execute the subroutine *Err_EventHistoryButton_Click* if an error condition occurs during the execution of the subroutine *EventHistoryButton_Click*. Anytime one function calls another, a parent/child relationship is formed between the calling and called routines. This relationship is noted in the *Parent-Child* table of the peeking database tables. One does **not** tie requirements to this statement. Requirements that are applicable to the calling of the function *Err_EventHistoryButton_Click* will be made when the code segment, *Err_EventHistoryButton_Click*, is examined.

```
Dim stDocName As String
Dim stLinkCriteria As String
stDocName = "Action Timestamp"
```

⇒Again, the above statements are simply data declarations and thus do not require any action.

```
If IsNull([Number]) Then
  MsgBox ("Cannot view history for record without an assigned Defect Number")
Else
   stLinkCriteria = "[number]=" & Me![Number]
   DoCmd.OpenForm stDocName, acFormDS, , stLinkCriteria
End If
```

⇒The above is a conditional expression that explicitly makes a decision regarding one of two options. The first conditional checks to see that a variable named *Number*, is not null. If it is, then an error message is given to the user as feedback. If it isn't null, then *Number* is used as a link criteria in order to open a form, namely the event history form. Essentially, there is a button on the screen that, when clicked, opens another form based on the key field *Number*.

⇒The actions taken from this conditional are to make an entry in the Code-To-Requirements peeking table that links requirements to code segments. There are really two requirements in the above conditional expression. The first requirement is associated with the act of providing specific feedback to the user when the variable *Number* is undefined, while the second is that a user has a method provided to them for viewing event histories.

```
Exit_EventHistoryButton_Click:
   Exit Sub
```

⇒The above subroutine is never explicitly called and has no real actions associated with it other than to exit this routine. As such, no peeking database entries are required.

```
Err_EventHistoryButton_Click:
   MsgBox Err.Description
```

Resume Exit_EventHistoryButton_Click

⇒As seen previously, this subroutine can be called from within the routine *EventHistoryButton_Click*. It does do something, namely it provides a generic error message to the user when any unexpected error condition is encountered. This is common programming practice but shouldn't be simply overlooked. It implies that the programmer wants to handle any unexpected circumstances and provide feedback. This has become a requirement of this system. It should be a general requirement to this application as well as any application and thus an entry is required in the Code-To-Requirements peeking table to establish this fact.

End Sub

⇒Merely the syntactically correct ending to the subroutine in Visual Basic. Requires no action.

11. Summary

The proper application of the Peeking Methodology to a project will significantly increase the number of requirements specified for that project. The increased requirement set will significantly reduce the susceptibility of having hidden requirements scattered throughout an application and thus will reduce the risk of there being defects discovered in those hidden requirements. Furthermore, the future risk associated with attempting to maintain de facto requirements is precipitously reduced.

Peeking has other far reaching consequences that make it worthy of consideration. The impact on regression testing can be significantly reduced by utilizing the linkages from source code, to requirements, and to test cases. The ability to increase the depth and accuracy of documentation for a product as well as maintaining accurate documentation also carries great potential.

In short, the Peeking Methodology has the potential to improve many areas of software development, testing, and documentation. The additional cost to an organization that already follows a life cycle model similar to Aztek's life cycle model is minimal.

12. Future Directions

There are several areas that could extend the Peeking Methodology. Each is examined below.

12.1 Tools to Aid in Requirement and Source Code Reuse

Requirements come in many flavors, with one such flavor being generic. Generic requirements can and should be easily adopted from project to project. If the requirements are tied to the source code, then the source code can easily be identified and possibly reused.

12.2 Extending the Technique to Cover Documentation Needs

Peeking offers potential rewards with respect to documentation requirements. User's guides are essentially a detailed explanation of the system requirements. Just as system test benefits by having a more complete understanding of system requirements, documentation also benefits. Although no attempt has been made to extend the peeking technique to documentation needs, one can easily envision documentation sections tied to code segments and thus requirements.

12.3 Smart Editors

Smart editors that could take advantage of the peeking tables could dramatically improve coding techniques as well as reduce unexpected faults incurred during the software maintenance portion of a product life cycle. Imaging a developer who, while viewing a particular function within a large set of application code, could just as easily view the requirements that were associated with this function.

13. Acknowledgments

It is with my children in mind that I realize one must take risks—to step out of the common path and to try a different route for if we all are to tread along the same path, our children shall never benefit from having us go first. It is with this in mind that I dedicate my energies to my children.

I also give special thanks to the summer student, Matt Dew, who actually wrote the AzTrack application and participated in the study.

Lastly, I once again thank my editor, Bernice Volinsky, who continues to have the energy and patience to correct my grammatical mistakes.

14. Trademarks

All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders.

15. References

[I] C. Mazza, J. FairClough, B. Melton, D. De Pablo, A. Scheffer, and R. Stevens, Software Engineering Standards, European Space Agency, 1994, Prentice Hall, pp. 8-11, ISBN 0-13-106568-8.

[II] L. Little, Taking a Peek Inside the Black Box, Proceedings of the Sixth International Conference On Software Testing Analysis & Review, May 1997, pp 530-542.

View Slides

Atoms and Bits, Pencils, Word Processors and Quality

Brad J. Cox, Ph.D., *George Mason University Program on Social and Organizational Learning*

Abstract

By creating a robust basis for buying, selling, and owning goods made of bits, superdistribution could put software engineering onto the same growth curve that hardware engineering has occupied since the industrial revolution. By providing a two-tier infrastructure for enforcing ownership of digital property, software engineers could assemble their products just as hardware engineers assemble theirs, building upon other people's efforts instead of fabricating everything from first principles.

Dr. Brad Cox *is a faculty member of the George Mason University Program on Social and Organizational Learning (PSOL). He has authored several books:* Object-oriented Programming, An Evolutionary Approach, *and* Superdistribution: Objects as Property on the Electronic Frontier. *He founded the Coalition for Electronic Markets whose objective is to build and deploy a nationwide revenue collection infrastructure for commerce in electronic goods. He has worked in industry using Artificial Intelligence and object technology and originated the Objective-C™ programming language. He received his Ph.D. from the University of Chicago in neurophysiology, now called neural networks.*

URL: http://www.virtualschool.edu/mon

STATUS REPORT
NEW LAWS THAT WILL GOVERN SOFTWARE QUALITY

Copyright © Cem Kaner, 1997. All Rights Reserved.

Abstract

The law of software quality is a complex area including laws governing crimes, negligence, fraud, deceptive advertising, unfair trade practices, unfair competition, anti-competitive practices, safety regulations, breach of contract, and various other areas. Legislators are now focusing more directly on computer-specific and software specific issues. One example is a proposed set of revisions to the Uniform Commercial Code that brings a wide range of issues into a single huge (340 page) statute that will govern almost all contracts for the development, sale, licensing and support of software. Another example is the wave of digital signature laws, laws that validate a computer-mediated technology in order to make possible a new way of doing computer-mediated business. These laws determine baseline standards for our products and services. As advocates of software quality, we should have an interest in anything that sets minimum software-related quality standards. Along with an interest, we can play an important role, because we understand the technical issues and the technological risk management issues much better than the attorneys who are trying to draft these laws. Our expertise has value for the legislative drafting process, and the absence of our expertise has resulted in draft laws that are weaker than they could be.

About Cem Kaner

I teach software testing courses, consult to software companies on testing, documentation and software project management, and practice law within the software community. I work to improve software customer satisfaction and safety *and* corporate profitability.

As a software developer, I've programmed, done UI design, managed software development projects, software test groups and documentation groups, sold software, and consulted to companies to build or rebuild project teams or small departments. I'm senior author of the book, *Testing Computer Software*, and am writing *Bad Software: A Consumer Protection Guide.*

As an attorney, I typically represent individuals and small businesses, whether they are developers or customers. Most clients are involved with software or with writing. I draft and negotiate contracts and advise clients on their rights after a breach of contract or a failure of a relationship. And I file expensive bug reports. I also do extensive legislative work, participating in multi-year projects that are drafting laws to govern software contracting and software-related aspects of electronic commerce. These include *the National Conference of Commissioners on Uniform State Laws'* (NCCUSL's) drafting committee *for Article 2B of the Uniform Commercial Code (Law of Licensing)*, NCCUSL's drafting committee for a *Uniform Electronic Transactions Act*, and the *Department of State's Advisory Committee on Private International Law: Study Group on Electronic Commerce.* (I am an actively participating "observer" at the NCCUSL meetings, not one of the 300 NCCUSL Commissioners.) I've also served as a Deputy District Attorney (full-time *pro bono* assignment) and as an investigator/mediator for a consumer protection agency.

I hold a Ph.D. in Experimental Psychology, with a focus on human perception and performance. I apply this in my work in Human Factors (how to redesign systems and machines so that they work better for people.) I hold a B.A. in Arts & Sciences (Math, Philosophy), and a J.D. (law degree). I'm Certified by the American Society for Quality in Quality Engineering.

STATUS REPORT
NEW LAWS THAT WILL GOVERN SOFTWARE QUALITY

Cem Kaner, J.D., Ph.D.

Laws that will govern the technical decisions that we make are being written by lawyers who, in the main, have little appreciation of the underlying technologies they seek to govern. The primary inputs to the drafting committees come from attorneys who represent the larger companies in the industry or the main trade associations. (The *professional* associations are not sending lawyers, just the *trade* associations.)

These lawyers have a keen appreciation of commercial risks and a sense that you can manage these risks by appropriate contracting and legislation. For example, if companies can be sued for defective products and your company doesn't like facing lawsuits, you can manage the risk of lawsuits *technologically* by making better products or by advertising them more honestly. Or you can manage the risk *commercially* by drafting contracts and laws that make it harder for your customers to sue you.

- In the extreme, the effect of a technological risk management strategy is a system that imposes huge direct and indirect taxes on us all in order to develop products that will protect fools from their own recklessness.
- In the extreme, the effect of a commercial risk management strategy is a system that is indifferent to quality, so long as the more powerful person or corporation in the contract is protected if the quality is bad.

Both strategies are valid and both are important in modern technology-related commerce. And both present characteristic risks. Unfortunately, the current fashion in legislation is an almost exclusive focus on commercial risk management. Part of the problem is that so few of us on the technology side are making ourselves available to explain technological risk management to these commercial lawyers. If the only tools that lawyers have are hammers, they will pass laws declaring that all non-hammers are nails.

This session looks at three clusters of legislative efforts, not from the point of view of how good the proposed laws are, but from the point of view of encouraging you to think about how you can be involved in constructive improvement to these laws, whatever your political philosophy. The three clusters include software contracting, the professionalization of software quality engineering, and electronic commerce.

The Law of Software Contracting

Attempts to create a unified law of software-related contracting are finally near completion after about ten years' work, in the American Bar Association as a model law for software licensing and then in the National Conference of Commissioners on Uniform State Laws as a multi-state-government-funded effort to create a software law that will be adopted in all 50 of the United States.

We now have a huge (340-page) proposed addition to the Uniform Commercial Code (Article 2B: The law of licensing). It is scheduled for introduction into state legislatures in September, 1998.

As far as I can tell, I was the first advocate for small customers (consumers and small businesses) to get involved in this work, and I started only 21 months ago. The law is intensely biased against customers, to the extent that Brian Lawrence and I cautioned readers of an *IEEE Software* paper that if we software development folk don't get involved, we'll have a product that may as well have been written by Dilbert's boss and lawyers who work for him.

You can read my analyses at my web site, which will probably be `www.badsoftware.com` by then. If that address still doesn't work, you can link to the site under a temporary name by going to my technical consulting site, `www.kaner.com`.

In this talk, I'll explore three issues as examples of the need for stronger technical input. I think that you'll see these as problems no matter what you think of the overall philosophical approach of Article 2B:

Liability for viruses

Article 2B (Section 2B-313, *Electronic Viruses*) imposes a duty on software publishers to exercise reasonable care to keep viruses off the disks they sell. The statute then defines the duty. Publishers are required to use *one* virus checker to check for *known* viruses. Publishers of software that costs $500 or more, that is sold under a site license, or that is delivered over the Internet are not required to use a virus checker as long as their license agreement (the one you see after the sale) says that they didn't necessarily test for viruses. The statute also requires customers to check software that they buy for viruses. The customer who doesn't run the test can't collect any damages if her system is trashed by a virus on the publisher's disk. The customer who does check for viruses and finds one, or who gets a virus from the publisher's product after being prevented from checking for viruses by some characteristic of the publisher's product or instructions, is entitled to a refund (on return of the product) if the publisher's product does carry a virus.

1. To my eyes, the standard of care (one virus checker, known viruses) is absurd. What is "standard practice" and how does it vary across platforms?
2. To my eyes, the concern expressed by publishers that they can't control what happens in manufacturing or in delivery over the Net is absurd. Am I wrong that we can safely control and check against manufacturing masters today? What about secure delivery over the Net? Where are we on this and how practical is it for a small publisher? Should we have a blanket exemption for publishers who posted a clean product but the product as delivered (after interception) was virused? Or should we plan for more secure delivery systems?
3. Do we have quotable examples that we can provide to the drafters as models of what goes right or wrong?

Embedded Software

Article 2B leaves to Article 2 (the Law of Sales) all issues involving quality of embedded software. Article 2 provides slightly stricter quality standards than 2B, and I have often been told that embedded software is, on average, developed under looser standards than shrink-wrapped software. If so, embedded software manufacturers will want to move their products over, so that they are covered by 2B, not 2.

How do we tell the difference between embedded and non-embedded software? I don't know. How, then, should this statute guide judges in deciding on a case-by-case basis whether the trial should proceed under Article 2B or Article 2? I don't know. I'm preparing a separate paper on this, for distribution to the Article 2B and 2 drafting committees. The first version is based on extended discussion in the `swtest-discuss` mailing list. That draft may then circulate to `comp.software.testing` and `comp.software-eng` newsgroups for additional discussion. Alternative memos to guide the committee will be very helpful.

Here is Article 2B's current attempt to distinguish embedded from non-embedded. Embedded software includes:

> `a sale or lease of a copy of a computer program that was not developed specifically for a particular transaction and which is embedded in goods other than a copy of the program or an information processing`

machine, unless the program was the subject of a separate license with the buyer or lessee.

Definition of defect

Under most circumstances, a defect in a product has to be very serious ("material breach of contract") or the customer will not be entitled to a refund for the defect. Initially, the statute defined a material breach almost exclusively in publishers' terms, using examples like failure to make payments for the software or using the product in a way that interferes with the publisher's intellectual property rights. I pushed heavily for a more customer-focused and product-focused definition (Kaner, 1997a) and made minor headway.

SECTION 2B-108. BREACH OF CONTRACT.

(a) Whether a party is in breach of contract is determined by the contract. Breach of contract includes a party's failure to perform an obligation in a timely manner, repudiation of a contract, or exceeding a contractual limitation on the use of information.

(b) A breach of contract is material if the contact so provides. In the absence of an express contractual term, a breach is material if the circumstances, including the language of the agreement, reasonable expectations of the parties, standards and practices of the trade or industry, and character of the breach, indicate that:

(1) the breach caused or may cause substantial harm to the aggrieved party including imposing costs that significantly exceed the contract value; or

(2) the breach will substantially deprive the aggrieved party of a benefit it reasonably expected under the contract.

(c) A material breach of contract occurs if the cumulative effect of nonmaterial breaches by the same party satisfies the standards for materiality.

(d) If there is a breach of contract, whether or not material, the aggrieved party is entitled to the remedies provided for in the agreement and this article.

Uniform Law Source: *Restatement (Second) Contracts* § 241.

The reference to the Restatement of Contracts is interesting because it uses a different set of criteria that I prefer. As cited in the Article 2B draft comments following 2B-108:

The *Restatement (Second) of Contracts* lists five circumstances as significant:

(1) the extent to which the injured party will be deprived of the benefit he or she reasonably expected;

(2) the extent to which the injured party can be adequately compensated for the benefit of which he will be deprived;

(3) the extent to which the party failing to perform or to offer to perform will suffer forfeiture;

(4) the likelihood that the party failing to perform or to offer to perform will cure the failure, taking into account all the circumstances, including any reasonable assurances; and

(5) the extent to which the behavior of the party failing to perform or to offer to perform comports with standards of good

faith and fair dealing. *Restatement (Second) of Contracts* § 241 (1981).

Let me illustrate the difference. Suppose that you buy BugWare 2.0 from ShipIt Software. BugWare is a shrink-wrapped product, costing $100. They sold 100,000 copies. The program has several annoyances, but nothing so serious that it destroys your hard disk. You are dissatisfied with the product, you can prove that it has several defects, and you want a refund. Can you get one?

Under Article 2B, probably not. The "contract" is the shrink-wrapped piece of paper that comes inside the box. (Article 2B will validate the terms of these "licenses," including terms that would not be valid under current law.) So this contract, written by the publisher, will probably not define "material breach" in a way that includes misbehavior of the program. So, you have to prove that you've been substantially harmed or that the program doesn't provide the benefits that you reasonably expected. Maybe you can make this argument, maybe not.

Under the *Restatement of Contracts*, which 2B lists as its source for the material breach standard, you'd have an easier argument.

1. You can still prove that you've been deprived of benefit, but this is just one factor.
2. The next issue is important and totally missing from 2B. You are entitled to compensation for the benefit of which you're being deprived (yes, yes, Article 2B says it gives this too, *but the publisher can charge you $35 per call to ask for a $10 partial refund*.) Under the Restatement, if you have no realistic means of being reasonably compensated, that's a factor in determining that you're entitled to a full refund. Under 2B, it's not.
3. The Restatement's third factor, forfeiture, favors the mass-market customer. You suffer a forfeiture if I string you along for a year on a project and then at the end of the year I say, "Sorry, I don't like what you're doing, I'm not going to pay you. Go away." My rejection of your product is a rejection of all compensation for the work you did in developing the product. This issue simply doesn't arise in mass-market software. My rejection of one copy of a mass-market product involves just one transaction of many. You make or lose your investment based on a more general level of success in the marketplace, not on my one purchase.
4. The Restatement's fourth factor is also irrelevant in Article 2B but not to customers. If the publisher won't give you a bug fix for a mass-market product, 2B says too bad (publishers of non-mass-market products have a duty to attempt to "cure" defects, but 2B drops the duty for mass-market products). The Restatement says that the court should take into account the publisher's willingness to give a bug fix in determining whether the refund is called for or not.
5. And finally, the Restatement asks the judge to consider the publisher's good faith. You get to bring in all those advertisements and argue that even if they aren't enforceable warranties, they are evidence that the publisher is playing fast and loose with facts. You get to bring in any evidence that the publisher knew about the bugs you're complaining about before it sold you the product. Etc.

The mass-market customer is treated reasonably well under the Restatement, and badly under 2B.

The buyer of custom software should probably be treated differently. If you develop custom code for a wild and crazy customer, you might end up including some genuine atrocities in the product's user interface. (*You really want us to design it this way? Are you really, really sure? OK, you're the customer.)* If you follow the customer's instructions, the customer shouldn't be able to demand a refund if the instructions were stupid.

I tried to sort this out in Kaner (1997a), suggesting different ways of classifying program misbehavior that depended on the relationship between the developer, the customer, and the specification (if one existed).

I think that we need something to help guide judges determine what is a serious defect, under what circumstances, or we'll see random standards across states and countries. Issues of the amount of harm suffered by the customer and the extent to which the customer is deprived of benefit are so vague in the law that it will take years and years of litigation before *some* of us finally understand what they mean (the rest of us lawyers will still *not* understand).

My proposal went like this:

> ***Reflecting the Relationships Between Licensor and Licensee***
>
> The licensor is analogous to the seller in a traditional sale. Under Article 2B, what is sold in the typical software transaction is a license to use the software, rather than a copy of the software. The licensee is the customer, who buys the license.
>
> I think there are four common classes of software transaction:
>
> 1. ***The customer writes the specifications and requirements and asks the developer to write a program as specified.***
>
>   In my view, the software developer meets its obligations if it writes a program that meets the specifications. If the specs say "2+2=5" then the program does not breach the software contract if it generates the wrong answer (5) whenever it adds 2+2. *Let the specifier beware.*
>
> 2. ***The developer writes the specifications and requirements, in preparation for custom development, but the customer is sophisticated.***
>
>   If the customer is a computer expert, then it is able to review the requirements and specifications just as well as the staff of the developer. The customer is also probably in a better position to understand its own requirements than the developer. Therefore it is reasonable to hold this customer accountable for reviewing the specifications.
>
>   Article 2 of the current Uniform Commercial Code defines a "merchant" as follows:
>
>   > 2-104 (1) "Merchant" means a person who deals in goods of the kind or otherwise by his occupation holds himself out as having knowledge or skill peculiar to the practices or goods involved in the transaction or to whom such knowledge or skill may be attributed by his employment of an agent or broker or other intermediary who by his occupation holds himself out as having such knowledge or skill.
>   >
>   > 2-104(3) "Between merchants" means in any transaction with respect to which both parties are chargeable with the knowledge or skill of merchants.
>
>   Article 2B uses essentially the same definition:
>
>   > 2B-102 (26) "Merchant" means a person that deals in information of the kind, a person that by occupation purports to have knowledge or skill peculiar to the practices or information involved in the transaction, or a person to which knowledge or skill may be attributed by the person's employment of an agent or broker or other intermediary that purports to have the knowledge or skill.
>
>   When Microsoft buys Apple computers, it is a merchant. When a large, local hospital buys a bunch of Apples, it might be a big business, but it is not a merchant.
>
> 3. ***The developer writes the specifications and requirements, in preparation for custom development, but the customer is not sophisticated.***
>
>   When doctors, dentists, insurance brokers, small grocery store owners, and other small business people buy software, they have no clue how to specify the software, no clue how to evaluate a requirements document, no clue how to test the software, and no clue how to cost-

effectively find a consultant who has these skills. In common computer parlance, these customers are called *Clueless*.

In UCC parlance, these customers are *non-merchants*.

These customers rely on the knowledge and experience of the developer. If the developer makes errors in defining the requirements or the specifications, which result in serious errors in the operation of the program, this is the developer's bug, not the customer's.

4. ***The developer writes a mass-market product. The customer has no input into the design or development of the product.***

   In the mass-market case, design errors belong to the developer, not the customer. Internal specifications that were used during development are largely irrelevant to the customer. The end product works in a reasonable way, as advertised and as documented, or it does not.

The Proposed Statutory Language

This proposal modifies Section 2B-108 of Article 2B. I am a novice at drafting statutes. The language will be cleaned up during the Article 2B review process. The proposal expresses my sense of the fundamental differences between these transactions.

Proposed SECTION 2B-108. BREACH OF CONTRACT.

(a) Whether a party is in breach of contract is determined by the terms of the agreement and by this article. Breach occurs if a party fails to perform an obligation timely or exceeds a contractual limitation.

(b) A breach of contract is material if the contract so provides. In the absence of express contractual terms, a breach is material if the circumstances, including the language of the agreement, expectations of the parties, and character of the breach, indicate that the breach caused or may cause substantial harm to the interests of the aggrieved party, that the injured party will be substantially deprived of the benefit it reasonably expected under the contract, or that the breach meets the conditions of subsection (c), (d), (e), (f) or (g).

(c) If the licensee provides the specification documents that are incorporated in the contract, then a breach is material if:

   (i) the software fails to perform in conformance with and in the time required by express performance standards or specifications;

   (ii) the software fails to perform in conformance with the specifications and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

   (iii) where the specifications are silent, the software's performance is unreasonable and it results in costs to the licensee that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable licensor would consider the software's performance to be unreasonable.

(d) If the contract is between merchants, and it contains specification documents, then a breach is material if:

   (i) the software fails to perform in conformance with and in the time required by express performance standards or specifications;

(ii) the software fails to perform in conformance with the specifications and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

(iii) where the specifications are silent, the software's performance is unreasonable and it results in costs to the licensee that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable licensor would consider the software's performance to be unreasonable.

(e) If the contract is not between merchants, and the licensor provides the specification documents that are incorporated in the contract, then a breach is material if:

(i) the software fails to perform in conformance with and in the time required by express performance standards or specifications;

(ii) the software fails to perform in conformance with the specifications and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

(iii) the software fails to perform in conformance with the end user documentation or other documentation delivered to the licensee and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

(iv) where the specifications and other documentation are silent, the software's performance is unreasonable and as a result, it either deprives the licensee of a significant benefit of the product or it results in costs to the customer that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable person would consider the software's performance to be unreasonable.

(f) If the contract is for a mass-market license, then a breach is material if:

(i) the software fails to perform in conformance with the end user documentation or other documentation delivered to the licensee and this failure either deprives the licensee of a significant benefit of the product or results in costs to the customer that exceed the price paid for the software;

(ii) where the documentation is silent, the software's performance is unreasonable and as a result, it either deprives the licensee of a significant benefit of the product or it results in costs to the licensee that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable person would consider the software's performance to be unreasonable.

(g) A material breach of contract occurs if the cumulative effect of nonmaterial breaches by the same party satisfies the standards for materiality.

If there is a breach of contract, whether or not material, the aggrieved party is entitled to the remedies provided for in this article and the agreement.

Whether you like this particular proposal or not, the point to recognize here is that if we want the courts to operate from a sound definition of software defect, we have to write it.

Professionalizing Software Development and Testing

Efforts to professionalize software developers and software quality advocates come and go. By "professionalize" I mean that the government will declare us members of a "profession," require us to go to school and take specific courses, require us to pass licensing exams, and subject us to a risk of malpractice lawsuits if we do something that someone thinks is incompetent. For example, there was an attempt to require all software developers in New Jersey to be licensed a few years ago. The bill failed, but another one like it will come up one of these days.

I sat on an ASQC committee that did the final drafting of the Body of Knowledge for the CSQE certification. Some of the people involved in that process expressed the belief that this certification was a step along the road to professionalization of Software Quality Engineers. My impression is that this is not an uncommon view in this community. This view rests on two interesting assumptions: that there *is* such a thing as Software Quality Engineering and that there *is* a shared (or shareable) body of knowledge that reflects a genuinely shared understanding of theory and practice in this field.

I'm not going to challenge those assumptions here (but see Kaner, 1996a,b for some discussion). But I do want to raise a little red flag.

As far as I know, there are no serious legislative efforts in the works today to professionalize software developers or software quality advocates. But as the Year 2000 hype accelerates, the press will pay even more attention to expensive software related problems. We already see serious bugs getting widespread press coverage. And the collapse of some large, expensive, software development projects for state and federal governments has gotten its share of bad press too. As the Year 2000 publicity--about the huge programming effort and national expense being required to fix a few bugs--gets even more public awareness, some bright politician will decide that it would be a great idea to license software developers and testers and make them liable for malpractice. If the publicity associated with programming failures is bad enough, the public clamor for politicians to "do something" to prevent this from ever happening again will be significant. And if *we* push the idea of professionalizing the field, the politicians might give us what we ask for, whether it makes sense or not.

As leading software quality advocates, I urge you to think carefully about your role in this. If we "professionalize" what will the effect be? Are we going to improve the technological management of the risks of bad software and bad software contracting this way? Or will we merely create a system that provides for commercial allocation of risks *onto* individuals (us) and away from the corporations that hire or contract with us and that make many of the demands that get the projects or products into trouble in the first place?

Electronic Commerce

I'm still learning about electronic commerce and encryption technology. I'll have more details at PNSQC.

The core problem is the need to be able to make binding, enforceable agreements in a worldwide marketplace that does business electronically. Digital signatures are being proposed as a big part of the solution for this, in state legislatures, among the United States and other federal governments, and at UNCITRAL (United Nations Commission on International Trade Relations).

You can digitally sign something by using your private key (part of the public/private pair in public key encryption). The digital signature laws all hope to make this signature binding. The concern that I have involves fraudulent use of the key by someone who has somehow obtained a copy of it from the legitimate owner. My concern is based on study of Article 2B's electronic contracting rules and the

American Bar Association's Science & Technology Section's *Digital Signature Guidelines*, attendance at meetings discussing the guidelines, and monitoring of the e-mail correspondence on a mailing list associated with the United States' Department of State's Advisory Committee Study Group on Electronic Commerce. Some state laws have been passed, which I have not yet read. (I don't get paid for any of this work and I pay my own expenses, which are considerable. That limits the time that I have available for going through this material.)

Suppose that someone gets access to your computer, that your key is stored on your computer, and that they are able to copy enough of your system that they can crack the (relatively simple, because a human has to type it) password that you use to access your key. They can now pretend to be you, and they can run up transactions in your name. The sellers that they contact will check the validity of your key by contacting a Certification Authority (a private company that you register with). The Contracting Authority (CA) will say that the key that has been used is indeed the key associated with your name, and that you have not yet contacted them to repudiate (cancel) the key. The seller then ships merchandise to the crook, thinking that person is you.

If you used a MasterCard (or any of the other main credit cards) instead of a digital signature, you'd enjoy several protections. First, your liability for fraudulent use of the card is limited. MasterCard acts as an insurer. Second, MasterCard imposes several additional security restrictions, such as refusing to authorize delivery of some merchandise to any address other than your billing address, such as requiring merchants to phone MasterCard for authorization, such as limiting your credit limit so that your card can only run up a limited amount of credit, and so on.

Unfortunately, if you use a digital signature, you have none of those safeguards. There is no automatic limit on your losses from theft. There is also no mechanism for you to require sellers to contact you for authorization of large purchases, to set the equivalent of a credit limit that suspends the encryption key's validity as soon as the total spent while using it this month exceeds a limit, to restrict keys to specific uses (such as, only accept this as a non-monetary signature on court documents, or only accept this for purchases under $100 or for purchases that are of business-related merchandise) or to require sellers to ship only to your home address. I *think* that several of these issues could be solved technologically, by generalizing the format of the Certification Authority's standard verification record, in a way that allows key owners to specify terms and conditions under which sellers can rely on their key, and to include those as part of the record that is sent to the prospective seller. **It *appears* as though there are often hundreds of pages of terms and conditions that get associated with a digital signature, protecting the seller and the Certification Authority and, *apparently*, a host of other people--<u>everybody but the customer who will be subject to *infinite liability*</u> (lose the house, the dog, the bank account, go bankrupt, the works) <u>if someone obtains access to their key.</u>**

I have several examples of ways in which companies gain access to consumers' hard disks under circumstances that look like normal, harmless business transactions. The consumer would have no idea that her key had been copied, and would likely not discover that it was being used fraudulently for days or weeks.

So, I *think* that I'm looking at a design bug in this system that provides risk management to everyone but the person who needs it most (the customer who will be bankrupted if her key is compromised). Design bugs like this cost more to fix the longer they're allowed to stay in the system. There's benefit in making noise about these early, before the system is fully implemented around the world.

Someone recently asked me, "Who died and made you a software legislation tester?" I didn't interpret this as a friendly question, under the circumstances, but I liked the title. *Software Legislation Tester*. The world needs a few more of us. Many of you are capable of this. What's our next step?

References

American Bar Association, Section of Science & Technology, *Digital Signature Guidelines.* First sections available free at www.abanet.org

Kaner, C. (1996a) "Software Negligence and testing coverage", *Proceedings of STAR 96* (Fifth International Conference on Software Testing, Analysis, and Review), Orlando, FL, May 16, 1996, p. 313. Available at `www.kaner.com` and/or at `www.badsoftware.com`.

Kaner, C. (1996b) "Computer Malpractice", *Software QA,* Volume 3, #4, p. 23. Available at `www.kaner.com` and/or at `www.badsoftware.com`.

Kaner, C. (1997a) What is a Serious Bug? Defining a "Material Breach" of a Software License Agreement. (unpublished). *Meeting of the NCCUSL Article 2B Drafting Committee*, Redwood City, CA, January 10-12, 1997. (abbreviated version, *Software QA, 3*, #6.) Available at `www.kaner.com` and/or at `www.badsoftware.com`.

Kaner, C. & Lawrence, B. (1997, March/April) "UCC changes pose problems for developers", *IEEE Software*, available at `www.computer.org`.

National Conference of Commissioners on Uniform State Laws (1997) *Draft: Uniform Commercial Code Article 2B – Licenses: With Prefatory Note and Comments.* Available at `www.law.upenn.edu/library/ulc/ulc.htm` in the Article 2B section, under the name *1997 Annual Meeting*. By the time you read this, a later version will also be available and posted at that site or at `www.law.uh.edu`.

Trustworthy Software for Today and Tomorrow

Lawrence Bernstein

President of National Software Council

View Slides

Do you lose data when your software system crashes and comes back up again? Too often the answer is 'yes.' Reliable software behavior must be achieved as people come to depend on systems and networks for their livelihood and their very lives. Software is the key to these systems, yet it is rarely discussed. It is the technology base we build on, but it has a weak theoretical foundation.

Dynamics

Most of the existing software theory focuses on its static behavior, that is an analysis of the source listing. There is little theory on its dynamic behavior, that is, how it performs under load. So we try to avoid serious software problems by over-engineering and over-testing. Often we do not know what load to expect. Let's listen to Dr. Vinton Cerf, inventor of INTERNET, "applications have no idea of what they will need in network resources when they are installed."

Today's software technology cannot support the scalability; robustness and reliability needed to manage heterogeneous networks. In their article on scalable software libraries, Don Batory and his colleagues at the University of Texas at Austin argue that a large, feature-rich collection of software components is inherently unscalable [Batory93]. They conclude:

"Contemporary software (template) libraries are populated with families of data structure components that implement the same abstraction. Each component is unique in that it implements a distinct combination of data structure 'features' (e.g., type of data structure, storage management, and concurrency). Every component is written by hand and utilities that are shared by many components are factored into separate modules to minimize gross code replication. We have argued that this method of library construction is inherently unscalable. Every time a new feature is added, the number of components in the library doubles. The number of data structure 'features' that one finds in today's libraries is woefully inadequate to address the needs of most applications; the data structures found in operating systems, compilers, and database systems are far more complex than those available in today's libraries."

Software engineers can not make sure that a small change in software produces only a small change in system performance. Industry practice is to test and retest every time any change is made. Those shops that ignore this practice suffer system crashes and cranky customers. The April 25, 1994 Forbes [Ross94] points out that a three-line change to a 2-million line program caused multiple failures due to a single fault. Software failures, not faults, need to be measured. Software stability needs study so that design constraints can be found which will assure reliable performance. Instabilities can arise when:

1.Computations cannot be completed before new data arrives.
2. Roundoff errors or buffer usage builds and eventually dominates system performance.
3. The algorithm embodied in the software is inherently flawed.

Software Rejuvenation

The first constraint needed to make systems trustworthy is to limit the state space in the execution domain. Today's software runs non-periodically allowing internal states to go without bound. Software Rejuvenation is a new technology that's limits the execution domain to being periodic. It gracefully terminates an application and immediately restarts it at a known, clean, internal state. It precedes failure, anticipates it and avoids it. It transforms non-stationary, random processes into stationary ones. One way to describe this is to stop running a system for one year, with all the mysteries that untried time expanses can harbor and run it one day, 364 times. The software states are re-initialized each day, process by process, while the system continues to operate. Increasing the rejuvenation rate reduces the cost of downtime but increase overhead. One system collecting on-line billing data operated for two years of operation with no outages. Its rejuvenation interval is weakly.

At Bell Laboratories an experiment showed the benefits of rejuvenation. A 16,000 line C program with notoriously leaky memory failed after 52 iterations. Seven lines of rejuvenation code with the period set at 15 were added and the program ran flawlessly. Rejuvenation does not remove bugs-it avoids them.

Chaos Theory

These potential instabilities must be considered in the design of software systems. Software's unstable behavior is remarkably similar to the chaos theory [Gleick87].

My conjectures are:

1. A software system can be modeled by the growth equation of Chaos Theory.
2. A set of tests can be found to characterize the k of a system.
3. The change in k of a software system can be computed for the changes being made to a software system.
4. A new k can be constrained to be less than or equal to the original k to assure stable behavior over time or rejuvenation can be used to keep the system away from the execution time when it becomes unstable.
5. Exhaustive regression testing would no longer be mandatory to assure reliable operation for all software changes.

The Chaos Theory growth equation may be a predictive model for the long-term execution of a software system. Software developers "know" that their systems can experience unexpected strange behavior, including crashes or hangs, when small differences in its operation are observed. These may be the result of new data, execution

of code in new sequences or exhaustion of some computer resource such as buffer space, memory, hash function overflow space or processor time.

A classical problem leading to this unreliable behavior is caused by systems interrupts. When the smallest segment of new code is added to a system. Experienced software managers know that they must retest and reverify system operation through an exhaustive set of regression and new functional tests. This retesting limits the growth in software productivity in enhancements to existing systems and modules. When problems are found in reused modules it is difficult for people unfamiliar with them to fix them and often leads to redoing the software rather than reusing it.

When the output of a system can be large to an arbitrary change, it is conditionally stable. The condition for stability is that the software system be tested for a particular set of input data and computer configuration. Experienced software managers know to look for "what changed" when a system that has been reliably performing suddenly and catastrophically fails. When its output grows without bound for a small change in input or in the object code that implements the system, the system is unstable.

If we can prove that the growth equation:
$X(n+1) = k\ X(n)\ (1-X(n))$ reasonably models software behavior, we can predict the conditions when an instability will occur. We may able to find the "k" for the system. Perhaps a set of tests can be found as those used to determine the order of linear continuous electrical systems that determine the software's k. Now when we change the software we may be able to find its new k'. We then will want to make sure the $k' < k$, so as to reduce the chance of instability. Even better, we may be able to compute a system's k and use it to predict instabilities. We then could find a new k" which will assure stability. If making the k smaller is not practical, we can implement rejuvenation to limit system execution duration and keep it away from times when it will be unstable.

Risk of Inaction

The point is that feedback control theory must be the habit of software professionals if we are to insure trustworthy software systems. One way to do this is to constrain the dynamic behavior of the software system by having designers follow design rules. The problem is that we do not have the all rules we need. If we do not act we will continue to have reports like those showing that thee Voyager and Galaxies spacecraft software had 149 safety critical errors.

The IRS software mess headlined on the front page of the January 31,1997 issue of The New York Times happens too often. Software problems are typically the root cause of these failures. Since anyone can write software and claim to be software expert, software consumers have no objective way of choosing their suppliers. If the industry continues to resist adopting firm ethical standards of behavior, government must impose them. Safety critical software is as important to our health and welfare as drugs and transportation. Why is software never recalled?

The National Software Council, an independent software professional association, is studying this problem. Many think software professionals must be licensed. The public and customers need to insist that software suppliers accept a code of ethics. Customers must not buy software from unethical firms. The problem is that we have no widely accepted code and, without it, customers have no choice but to buy software from the lowest bidder and hope for the best. This too often leads the kinds of problems faced by the IRS. They are not alone in suffering from software malpractice. The long delay in opening the Denver Airport, the failure of the French missile Ariane 501, Patriot missile failures in Desert Storm and telephone carrier failures are a few of the legion of software bankruptcies.

At a minimum the software industry needs to adopt and software consumers need to insist on this modest code of behavior:

For each project, a software architect and project manager is identified by name. Both attest that the software is 'fit for use.' They analyze software project risks and document their findings. They make sure that user interfaces are intuitive and easy to use, that 'help' is helpful, that private information is protected and that the software is fit for humans. They understand the problem and don't just accept the customer's solution. They follow formal documented software development processes. They respect property, copyright, patent and privacy rights. They publicly advocate ethical behavior.

National Software Council

The National Software Council (see http// www.nscusa.org) is an independent organization designed to keep software strong and grow its contribution to national well being. One of its goals is to assure that trust in software is well founded. Another is to make the United States of America the model information society through the most efficient production and application of the most ethical, safe, secure, reliable and usable software.

The National Software Council believes that:

§ Computer software is the raw material of the information society.
§ Society's dependency on software is increasing daily. People do not know how many lines of code influence their lives nor who wrote them.
§ As software spreads, society becomes more vulnerable to the consequences of software failures.
§ The economic future and safety of societies depend on the mastery of software development and application.
§ Taxpayers, consumers and corporations are paying billions of dollars each years for systems that are never delivered, are unreliable, unsafe and are vulnerable to sabotage.
§ Government, industry and academia must unite to create a professional and ethical software industry.
§ Software development, performance and reliability are unpredictable with today's technology. Even so, current state-of-the-art processes, tools, methods and technology

need to be widely adopted. Work to advance the state-of-the-art needs to be funded and supported.
§ We need to charter impartial organizations to assess software systems for use by the public, in government or by industry.

Thinking globally, sharing nationally and acting locally is the way the NSC works.

Conclusion

Software professionals need to take responsibility for the performance of the systems they build. They need to adopt a code of ethics and study how the software performs under load. They must make earn the public's trust by making their systems trustworthy.

References

[Arth83] Arthur, Lowell Jay, Programmer Productivity: Myths, Methods, and Murphology A Guide for Managers, Analysts, and Programmers, John Wiley & Sons, New York, 1983, pp. 25-27.

[Batory93] Batory, D., Singhal, V., Sirkin, M., and Thomas, J., Scalable Software Libraries, Department of Computer Sciences, The University of Texas at Austin, SIGSOFT '93, December 1993.

[Boeh84] Boehm, Barry W., Gray, T. E., and Seewaldt, T. "Prototyping Versus Specifying: A Multiproject Experiment," IEEE Transactions on Software Engineering, Vol. SE-10, No. 3, May 1984.

[Des94] Desmond, John, "IBM's Workgroup Hides Repository," Application Development Trends, April 1994, p. 25.

[Dijk72] Dijkstra, E "The Humble Programmer," 1972 ACM Turing Award Lecture in Classics in Software Engineering, ISBN 0-917072-14-6, 1979

[Fair94] Fairley, Richard. "Risk Management for Software Projects," IEEE Software, May, 1994, pp. 57-67.

[Gleick87], Gleick James, Chaos Making a New Science, ISBN 0-670-81178-5, 1987, R. R. Donnelly and Son, Virginia (See especially page 71).

[Jone86] Jones, Capers. Programming Productivity, McGraw-Hill Book Company, New York, 1986, p. 83-210.

[Mill88] Mills, Harlan. Software Productivity, Dorset House Publishing, New York, 1988, pp. 13-18.

[Poul93] Poulin, J. S., Caruso, J. M. and Hancock, D. R. "The business case for software reuse," IBM Systems Journal, Vol. 32, No. 4, 1993, pp. 567-594.

[Pres94] Pressman, Roger. "Hackers in a Decade of Limits," American Programmer, January 1994, pp. 7-8.

[Ross94] Ross, Philip E., The Day the Software Crashed, Forbes, April 25, 1994, page 150.

[Selb88] Selby, R "Empirically Analyzing Software Reuse in a Production Environment. In Software Reuse:Emerging Technology, W. Tracz (Ed.), IEEE Computer Society Press, 1988 pp. 176-189.

[Wals77] Walston, C. E. and Felix, C. P. "A method of programming measurement and estimation," IBM Systems Journal, No. 1, 1977, pp. 54-60.

[Your79] Yourdon, Edward Nash. Classics in Software Engineering, Yourdon Press, New York, 1979, p. 122.

World Class Software Quality in Practice

George Yamamura and Gary B. Wigle

Boeing Information, Space & Defense Systems
P.O. Box 3999, MS 87-67
Seattle, Washington 98124-2499
Voice: (206) 773-3762
E-mail: george.yamamura@boeing.com
or gary.b.wigle@boeing.com

ABSTRACT

The Boeing Space Transportation Systems (STS) organization has a long history of process improvements with measured results and impressive performance and customer satisfaction, truly a real success story. The STS organization has been assessed using the Software Engineering Institute (SEI) Capability Maturity Model (CMM) at Level 5. This paper focuses on software quality as related to the organization's continuous improvements in software development and process activities. A clear road map for process improvement is defined by the STS management framework. As one proceeds along the process improvement road, there is a paradigm shift in the way the organization views its activities. The development of software goals at multiple levels that are traceable to business goals is a natural step. This paper includes Boeing STS data showing nearly 100% defect detection, a 1:7 ratio in cost:benefit of defect prevention, 240% improvement in software productivity, and millions of dollars in software underruns. The ultimate measure of success is still customer satisfaction that can be measured in terms of cost, schedule, software quality, and software performance. Boeing STS data is presented that shows excellent customer satisfaction as measured by management and performance indicators.

KEYWORDS

SEI, CMM, Level 5, Process Improvement

BIOGRAPHIES

George Yamamura is the Software Engineering Process Manager of Boeing's Information, Space & Defense Systems (ISDS) in Seattle, Washington. He managed the Space Transportation Systems (STS) software development organization and its efforts leading to a SEI CMM Level 5 rating. Mr. Yamamura has thirty years of software experience in the space application field. He is currently managing the ISDS software engineering process group supporting Boeing programs and working on deployment of best practices. He has a B.S. and a M.S. in Aeronautics and Astronautics from the University of Washington and a M.S. in Applied Mathematics from the University of Santa Clara.

Gary B. Wigle is the Senior Principal Engineer for software processes in Boeing's Information, Space & Defense Systems (ISDS) in Seattle, Washington. He has more than twenty years of experience in embedded software applications both in the Air Force and at Boeing. He was the Software Engineering Process Group (SEPG) lead for Space Transportation Systems (STS), as it accomplished a SEI CMM Level 5 rating. He has been a leader in the development of Boeing software standards and many other process improvement efforts. He has a B.S. in Physics from the U.S. Air Force Academy and a M.S. in Systems Management from the Air Force Institute of Technology.

© THE BOEING COMPANY

World Class Software Quality in Practice

by
George Yamamura and Gary B. Wigle
Boeing Information, Space & Defense Systems

The Boeing Space Transportation Systems (STS) organization is producing high quality software using certified Level 5 capabilities as defined by the Software Engineering Institute (SEI) Capability Maturity Model [SM] (CMM). STS focused on process improvement as a priority for many years and demonstrated a high level of maturity through this formal assessment. The assessment was conducted using the CMM Based Appraisal for Internal Process Improvement (CBA IPI) method. This method is a rigorous approach requiring more evidence of institutionalization to be provided for all software development activities. The assessment was conducted over a ten day period with the team lead by Mark Paulk and Steve Masters of the SEI. This paper focuses on the history of process improvement in STS, the relationship to business goals, measured benefits, customer and employee satisfaction in a high maturity organization, and deployment of successful practices to leverage quality processes.

The Boeing STS organization consists of projects supporting space transportation for the Department of Defense (DoD) and National Aeronautics and Space Administration (NASA). These projects provide main launch systems and upper stage boosters for multiple satellite payloads. Each project has major embedded software required to interface with the on-board avionics for vehicle management and control. The software development functions include algorithm development, mission design, software-systems and requirements, design, test, product assurance and integration, and mission operations support.

More than fifteen years worth of process related data, captured in a process library that included more than one hundred notebooks, was reviewed during the assessment. This process data was summarized in a detailed presentation to the assessment team, followed by a further review of the data and interviews to confirm institutionalization.

In current assessments, emphasis is placed on institutionalization, which is a key concept to sustaining process improvement activities. One basic definition of institutionalization is having processes “documented, trained, practiced, and maintained”. Our experience shows that this definition does not go far enough. Process improvement will be sustained only when it is done for the right reasons - it is understood as a strategic part of your business goals. Additional attributes associated

[SM] The Capability Maturity Model (CMM) is a product of the Software Engineering Institute at Carnegie Mellon University

with institutionalization derived from our Level 5 STS organization include: "owned, believed, used with pride, and promoted". The processes are not just documented and practiced, but are truly owned by the engineers performing them. The engineers know that they are the recognized experts for their processes. When process measurement data and product performance demonstrate successful processes, the engineers believe in them even more and take pride in their use. Then they are in a position to promote these best practices for others to use. It is these attributes that are crucial for successful deployment to other projects. Because processes are institutionalized in this way, process improvement is sustained in the STS organization.

Process Improvement History

STS has a history of more than fifteen years of process improvement using several frameworks, as illustrated in figure 1. Long before the SEI CMM, STS had strong process champions motivated to manage with facts and data. In the early 1980's, the organization documented their software processes as a basis for improvement to eliminate future occurrences of defects. Also, a core set of metrics was defined to primarily measure status. STS participated with other Boeing organizations and the Central Software Engineering group to develop a company-wide embedded software development standard. Training in this standard was developed and made available to all software personnel. The Boeing software standard was more than 350 pages and

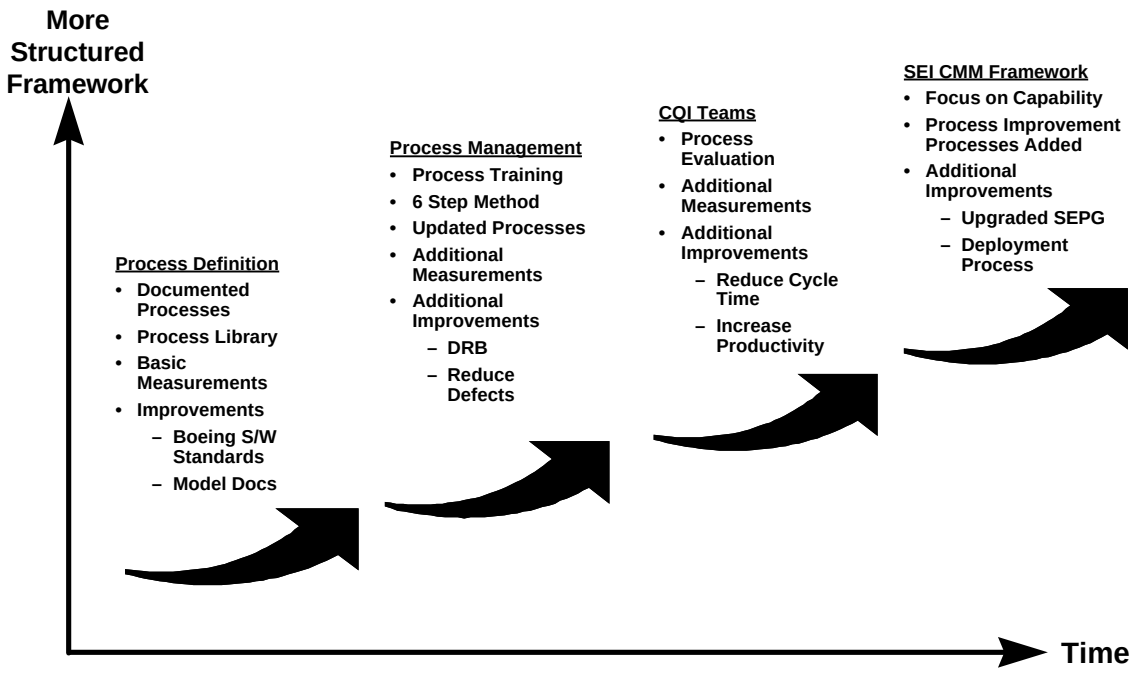


Figure 1 STS Has A Long History Of Process Improvement

defined the software development processes and associated products. Much of the input for this standard came from the STS organization. Although the STS organization was “grand-fathered” because its projects were already in place before the standard was implemented, STS used the standard as a basis to identify additional processes to capture.

In the late 1980's, an organization-wide process improvement effort was initiated using a six-step method for software. These steps included:

- Define the mission
- Define the process parameters
- Develop schedule for steps
- Develop process flow diagrams
- Define measurements
- Analyze, improve, and monitor

At this time, all documented processes were evaluated and updated accordingly. Defect data was used as a basis for instituting a formal design and code inspection process, which significantly reduced defects found beyond designer testing. A four step defect corrective action process was also implemented, reducing the recurrence of the most common errors. Employees took pride in their accomplishment of dramatically reducing defects.

In the early 1990's, continuous quality improvement (CQI) methods were implemented. CQI teams were formed and, again, all software processes were evaluated and updated accordingly. The focus of this effort was on increasing productivity and reducing cycle time, with 50% efficiency in cycle time reduction realized in some processes. The CQI methods provided a more formal structure for process improvement than the previous six-step method.

In the past two years, the SEI CMM has been used as a framework to support process improvement. This framework is specifically designed for software and provides a much more complete set of criteria to assess maturity. Most of the practices at Levels 2-5 were already institutionalized in the STS organization. The Software Engineering Process Group (SEPG) activities were expanded and formalized during this time, including responsibility to evaluate and implement new processes, and maintain the existing processes for the entire STS organization. Training, defect prevention, and technology insertion are also managed by the SEPG. Members of the SEPG include the STS software engineering manager, all project software managers and leads, key domain experts and the process focal points.

Figure 1 should not be viewed as a road map to Level 5 - that is not the intention. Figure 1 simply shows that process improvement has been occurring over many years and that the best practices were used during each period of time. Each framework

provided more structure and knowledge to the process improvement activities than the previous one. A project starting out today would use the SEI CMM as its framework for assessment, and achieve the same kind of improvements in a much shorter time period.

The Right Goals

A common practice that we see today is senior management initiating process improvement by establishing one of the SEI CMM levels as the goal, e.g., this organization will achieve SEI CMM Level 3 by June 1998. This approach has several pitfalls. First, it shows that senior management may not fully understand what the levels of the CMM truly represent. Second, the organization has not performed the initial self-assessment and planning to even know if such a goal is realistic. Third, senior management is not likely to provide the additional funding necessary for an aggressive approach like this. And fourth, if the organization is successful, they reach their goal and have no idea what to do next and the effort then collapses. A process improvement program must have the right goals to enable the effort to sustain itself.

As an organization matures through a process improvement program, there is a paradigm shift in the way people view their tasks. This paradigm shift can be described by four attributes that change focus as a project moves from CMM Levels 2 and 3 to Levels 4 and 5. First, at Levels 2 and 3, risk management remains a focus. That is, the organization is trying to improve its processes in order to minimize risk associated with software development. The initial efforts focus on basic management and engineering processes. As the organization moves to Levels 4 and 5, risk management is replaced by goal management. The organization understands its business goals and has derived software goals from them. Decisions are made with a full understanding of these goals. Everyone in the organization knows why they are doing what they do, and how a change to a task impacts the business goals.

Second, a Level 2 or 3 organization focuses on processes. The organization is trying to identify, capture, and understand the processes that are performed. This is a necessary first step to process improvement. In a Level 4 or 5 organization, these processes are well understood and process measurement data has been collected for some period of time. The processes are understood in quantitative terms. Now the focus shifts from defining and capturing processes to controlling software development by using these proven processes. The organization has data from which to estimate reliably, the ability to define variances for control, methods in place to track processes, and pre-defined procedures to perform when variances are exceeded. This control is performed by everyone in the organization, not only management.

Third, there is a practitioner focus in a Level 2 or 3 organization. Everyone is trained in process improvement. Then, everyone is asked to identify and help capture their existing processes. Next, members of the organization help improve these processes. The processes at Levels 2 and 3 are the basic management and engineering

processes necessary for software development. In a Level 4 or 5 organization, this practitioner focus shifts to a management focus. Software development is managed, processes are managed, and process improvement is managed. Quantitative techniques are used with well defined processes to managed all aspects of the organization. Engineers and managers have learned that they must manage their tasks in this kind of environment. This management focus is driven by organizational business goals.

And fourth, a Level 2 or 3 organization focuses on defining and capturing processes - to put them into practice and make them operational. But in a Level 4 or 5 organization, these processes are institutionalized (review the definition in the introductory section of this paper) and measurement data is used to continually improve them. We believe that the shift in focus in these four areas define the real difference between a low maturity organization and a high maturity organization.

We have talked about the right goals, software goals, and business goals and there may be some confusion about how they relate. Figure 2 shows the framework that we have established to derive the right goals (software goals) for process improvement. The organization starts by understanding what their business goals are. Figure 2 shows two business goals that are common to most organizations. The organization wants to be a leader in performance of its product, and have very satisfied customers in its current business base. In addition, the organization wants to be able to obtain more

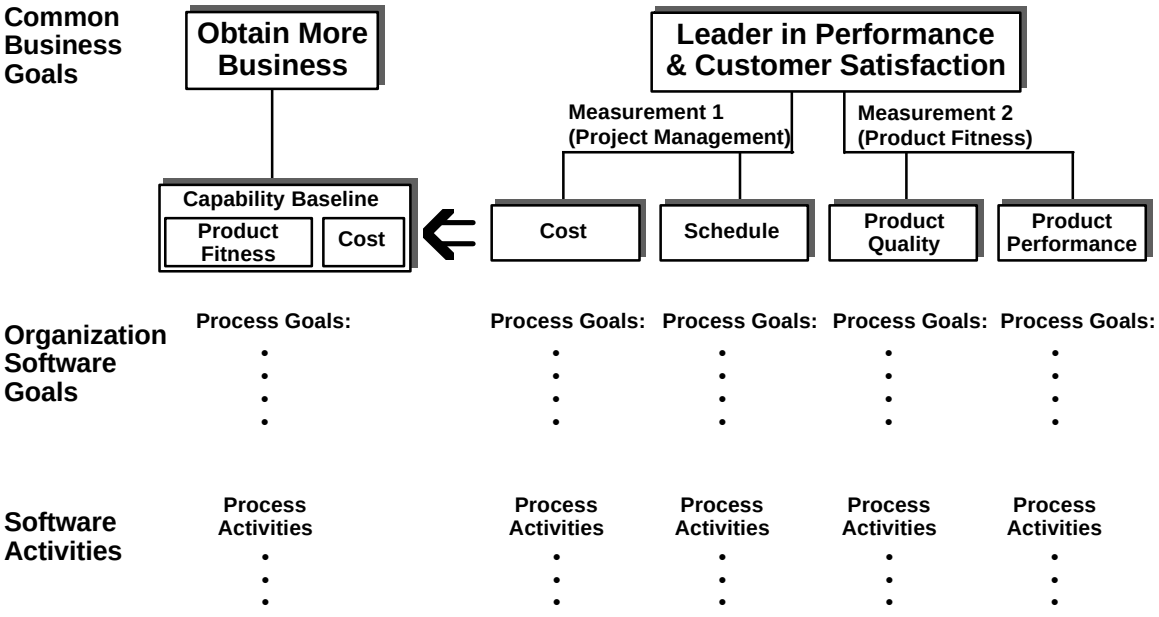


Figure 2 Software Goals Framework

business in order to stay viable. Both of these elements are crucial to long-term organizational success. Below these business goals are project indicators that have a direct bearing on the business goals. We have found that cost, schedule, product quality, and product performance are the key indicators for current contracts. How the organization performs on its existing contracts defines its capabilities, and these capabilities are what are used to estimate work for future business. In order to win new business, the organization must have the right product at the right cost.

The organization is ready at this point to analyze its processes and determine which processes have the most impact on each of the indicators. When these processes are identified, goals for each one can be established to support the business goals. The final step is to identify the software development activities associated with each of the key processes, and determine how they can affect the process goals. This is the level where real process improvement decisions are made - at the process level. But the decisions are made with a full understanding of the business goals of the organization. The software goals identified through this framework become the right goals for process improvement, and this kind of process improvement program based on business goals will be sustainable regardless of what SEI CMM level is achieved.

Measured Benefits

Software process improvement in the STS organization has resulted in defect reduction, increased productivity, launch call-up cycle time reduction, high product quality, excellent mission performance, high customer satisfaction, and very satisfied employees. In addition, a true cost benefit has been substantiated with facts and data. We have prepared nearly forty charts used in our presentations to show the measured benefits, based on the measurement data available in STS. Obviously, we cannot present much of that data in this paper. We have selected a few figures that will give you a sampling of the kind of benefits that STS has demonstrated by using high quality processes.

One measure of the effectiveness of the our processes is shown in figure 3. Defects found during the software development processes versus those found after delivery to system test were tracked. Initially, our processes were finding 89% of the defects; therefore, allowing 11% escapes. These defects contributed to the higher cost of testing and rework. With the improvements made to defect detection and prevention processes, the software processes now find nearly 100% of the defects, reducing the escapes to nearly zero. Further improvements are continuing within the software processes themselves to detect defects earlier.

Analysis of the cost to benefit ratio showed a definite cost savings by implementing process improvement, as shown in figure 4. For example, the formal inspection process improvement (the Design Review Board) increased total development effort by 4%. The black bars in figure 4 represent development costs, and the light gray bars represent the cost of rework associated with development. The dark gray bar reflects

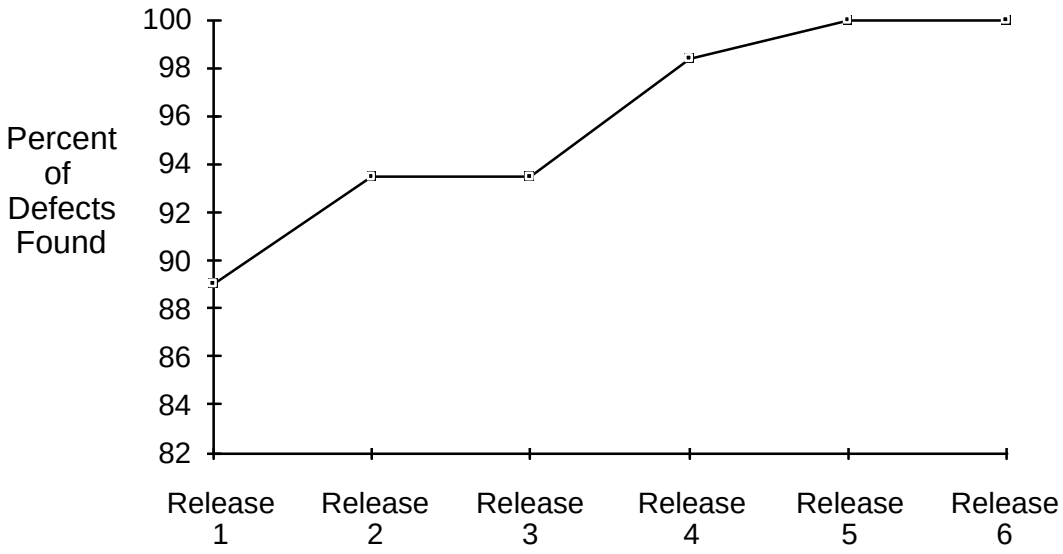
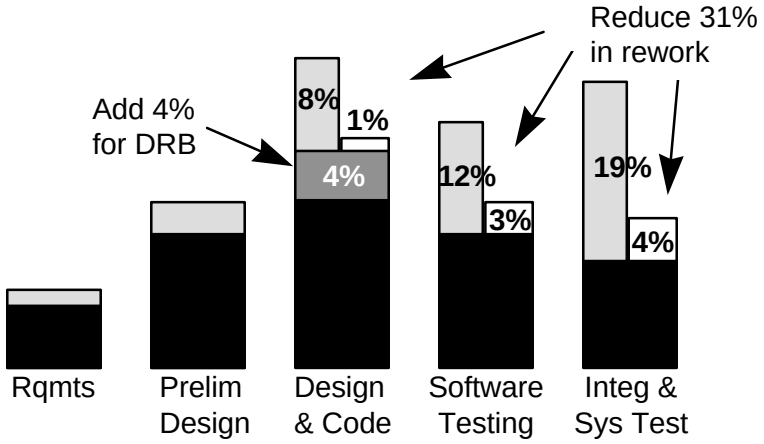


Figure 3 Defects Found Before Release



Implementing Design Review Board (DRB) increased development effort by 4%

Rework reduced in Design & Code from 64 to 7 defects	=	7%
Software Testing from 17 to 4 defects	=	9%
Integration & System Test from 10 to 2 defects	=	15%
		31%

Cost:Benefit ratio is 4% : 31% or 1 : 7.75

Figure 4 Inertial Upper Stage (IUS) Cost:Benefit Ratio

the increase in cost associated with the Design Review Board (DRB), while the white bars represent the rework after DRB implementation. You can see that DRB reduced the rework effort during development by 31%. Therefore, improving the inspection process which added 4% to the total development effort returned a 31% reduction in rework for a 1:7.75 cost to benefit ratio.

Our data also shows that software productivity increased with process improvement as shown in figure 5. We will not attempt to argue the definition of productivity, as many variations exist. The point is that however you measure productivity, you will see an increase as a result of implementing process improvement. Increased software productivity translated into software underruns and a return of funds to the customer. One such underrun is shown in figure 6, an underrun representing one and a half million dollars. This money did not represent cost avoidance, it represented money actually given back to the customer on an existing contract. Another underrun of nearly a third of a million dollars was returned to the customer on a later date. This sharing in realized benefits is one of the ways that the STS organization achieves unusually high customer satisfaction.

Customer and Employee Satisfaction

Customer satisfaction is tracked by two measurements in the STS organization (see figure 2). The first is an award fee, a subjective evaluation by the customer on program

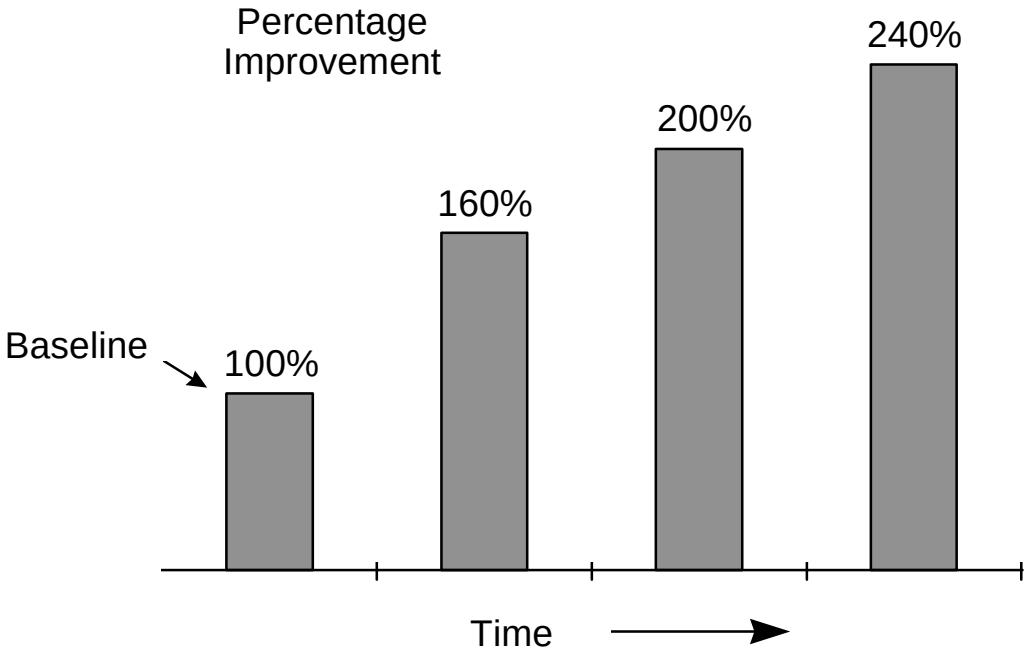


Figure 5 STS Software Productivity Over Time

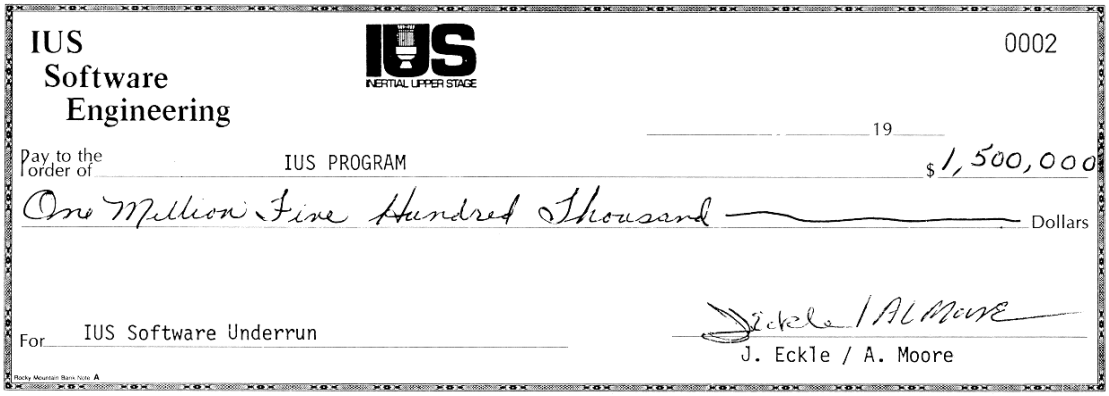


Figure 6 Software Cost Savings Resulting From Process Improvements

management performance (cost and schedule). This measurement is provided semi-annually by the customer. STS has earned a consistently excellent rating of 97-100% for more than six years.

The second measurement associated with customer satisfaction is an incentive fee. The incentive fee is based on the product's ability to perform the mission (software quality and product performance) and is provided by the customer after each launch. The operational performance of one of our products, the Inertial Upper Stage (IUS), is graphically illustrated by the bullseye chart in figure 7. The performance of the IUS is determined by how accurately it places its payload into position in three dimensional space. This accuracy is determined by how much propellant the payload must burn to get into final position after separating from the IUS. The bullseye represents the amount of satellite reaction control subsystem (RCS) fuel required to correct the final orbit. Incentive fee is paid based on this performance. No incentive fee is earned outside the bullseye, 25% is earned in the outer ring and so on, until 100% is earned in the center. A blow up of the target center shows that our performance has been in the inner 20% of the center of the bullseye. This is the kind of performance that the customer has come to expect from the Boeing STS organization.

Another of Boeing's STS software goals is to have a motivated and skilled workforce. This results in very satisfied and productive employees. A survey was taken before and after major process improvements. A comparison in figure 8 shows the degree of employee satisfaction. Before improvements, the development environment was ad hoc and somewhat chaotic with constant tiger teams and firefighting. The results showed 26% were dissatisfied with a mean near the neutral value. After process improvements and implementing a structured, disciplined development approach with clear goals, the satisfaction percentage jumped to 96% with a mean of 8.3 out of 10, showing a very satisfied workforce.

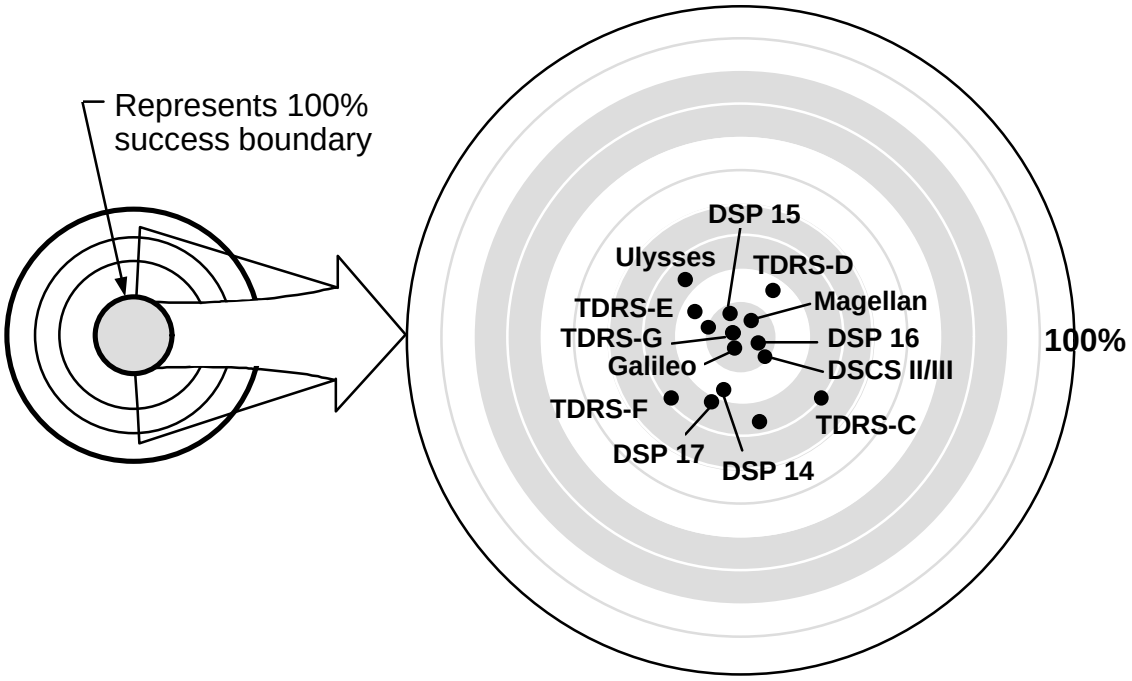


Figure 7 Performance Is In The Center Of The Bullseye

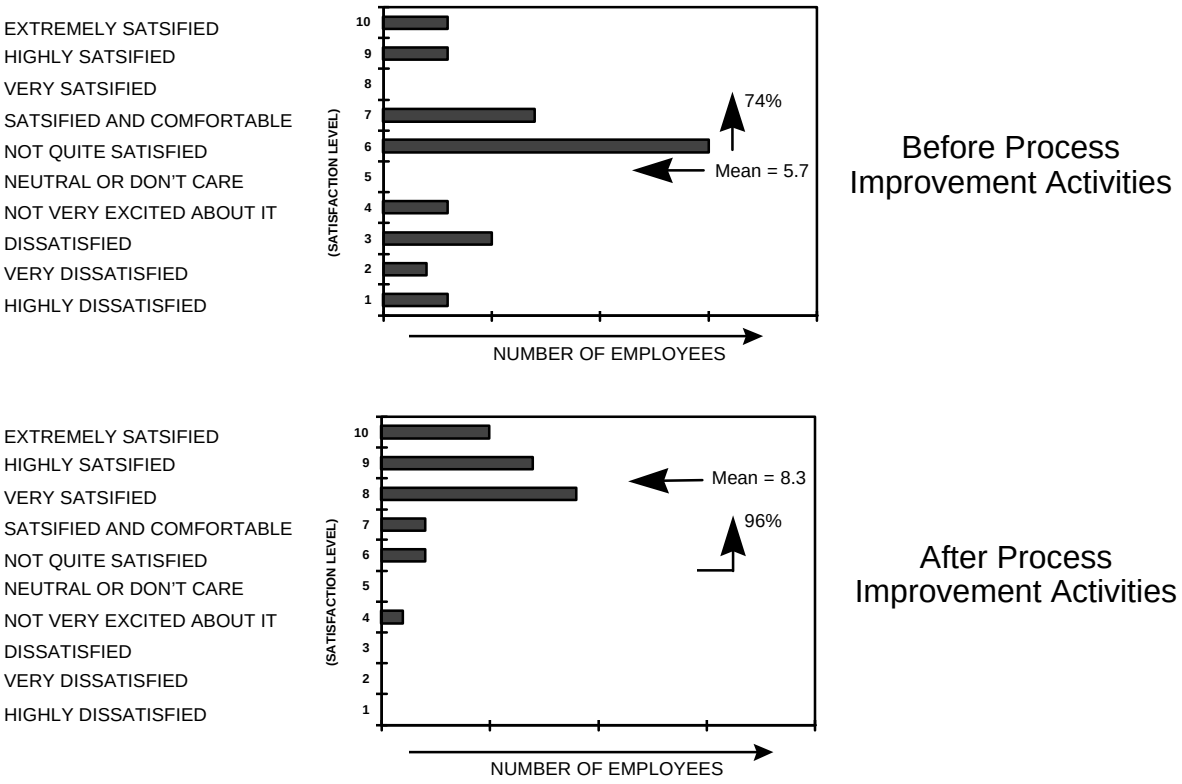


Figure 8 Employee Satisfaction

Deployment of Quality Practices

Having a strong process driven organization, the next challenge was to deploy the STS processes to new projects within the organization. The deployment process, like most STS processes has been developed and evolved over many years. The IUS project processes have been used on several other domain related projects (space launch systems). More recently, the processes were deployed to the Avionics Obsolescence Activity project and a New Launch Vehicle Start-up project. These projects were included in the assessment that yielded STS the Level 5 rating and demonstrated that STS could deploy its processes successfully to a start-up project less than two years old.

A brochure was developed that highlights the benefits of proven STS processes and identifies process assets that are available to new start-up projects. This brochure provides visibility of key resources that are available for immediate use within STS, along with contact points. For the deployment process to succeed, people need to know what is available. The brochure is one of the methods that we use to accomplish this objective.

We have had many discussions with people concerning what constitutes a successful deployment process. We believe that the deployment process has three key attributes. The first attribute is having a documented set of policies, plans, procedures, and templates that are immediately usable by new projects. The higher maturity that an organization has attained, the more comprehensive this set of process assets will be. The second attribute is having a pool of domain experts that know the processes and can be temporarily loaned to new projects to ensure speedy implementation. This attribute requires an organization to be willing to loan out key people without fear that the new project will take control of them. In some cases, we transfer a few key people to a new project and back fill into the existing projects to train others in the processes. Our IUS project has been coined “the University of IUS”. This way, there is a continuing pool of trained people to deploy. The third attribute is having key leads from a new project immediately designated and involved as active members of the organizational SEPG. This establishes process improvement as a visible function of the new project early in its development life cycle and demonstrates commitment. These three attributes are all necessary to a successful deployment process.

Although the STS deployment process is a general process, it does rely on domain (space launch systems) specific key resources. Because of this, full deployment of STS processes is limited to projects within STS. However, many of the STS key process assets can be used as models by other domain projects (fighters, bombers, helicopters, etc.) to develop key process assets applicable to their domains much faster. With this approach in mind, the authors have assumed responsibility for deployment of these processes to programs on a larger scale in Boeing Information, Space & Defense Systems (ISDS).

Process improvement begins from within; thus, a SEPG has been formed in the Central Software Engineering organization. The experiences gained in this activity will further help in understanding the needs of Boeing ISDS programs. In addition, an ISDS-wide SEPG has been formed to facilitate communication between programs, and to take ownership of the ISDS level process assets developed by Central Software Engineering and programs working together. The expectation will be that new programs begin with higher maturity practices in place. We have demonstrated that this goal is achievable.

Conclusions

Our organization implemented process improvement for the right reasons, to improve the big four (cost, schedule, quality and performance) and to support our marketing business. With the current SEI CMM emphasis, senior management too often establishes goals for achieving a specific level. This can lead an organization to lose sight of the real goals of process improvement.

The STS organization instituted process improvement because it was the right thing to do for the organization. Employees are motivated, the teams have ownership, significant improvements result, and processes are institutionalized and maintained. The benefits described in this paper were realized because our organization implemented process improvement for the right reasons. More importantly, everyone in the organization understands how their work supports the business goals, and how they can improve further. This understanding results in a highly effective and satisfying work environment throughout the organization.

10-piece Toolbox to get People to Change

Mary Sakry, *The Process Group*

Abstract

Many organizations try to implement change. This includes everything from the introduction of a new tool or method, to a company-wide process improvement program. Often these attempts fail due to a lack of skills to effect change. When new ideas are introduced they are either abandoned after a short time or adopted by only a few people. Whenever an organization wants to change there are some key principles that it must consider. This talk covers ten practical principles that lead to change.

Mary Sakry *of The Process Group specializes in software engineering process improvement. She has 21 years of software development, project management and software process improvement experience. She teaches and consults on process improvement, software inspections, SEPGs, SEI CMM, software project planning, estimating, and management. She is trained and authorized as a leader of the SEI SPA and CMM Based Appraisal for Internal Process Improvement (CBA IPI). Before life as a consultant, she worked both as a software designer and project manager at Texas Instruments prior to joining TI's corporate software engineering process group. She has an M.B.A. in Business Management, St. Edwards University, and a B.S. Computer Science, University of Minnesota.*

A Modular Software Process Mini-Assessment Method

View Slides

Karl E. Wiegers and Doris C. Sturzenberger

Eastman Kodak Company
901 Elmgrove Road
Rochester, NY 14653-5811
Phone: (716) 726-0979
Internet: kwiegers@kodak.com

Speaker: Karl E. Wiegers

ABSTRACT

A modular CMM-based software process mini-assessment method was developed to meet the diverse needs of software projects undertaking process improvement efforts at Eastman Kodak Company. This method consists of several well-defined components, each of which has two to four process options. During a planning meeting, the software project leader and the assessors select a specific set of component alternatives to construct a mini-assessment that will best satisfy the project's objectives within its time constraints. The flexibility provided by the method has been well received by participating projects.

The mini-assessment method is supported by an infrastructure including extensive procedural guidance for the assessors, and presentation slide modules that are used or adapted for the various meeting events. Templates, forms, checklists, and databases facilitate the creation of deliverables and help to ensure that a repeatable mini-assessment process is followed by all assessors. Summary results from the mini-assessments conducted to date are presented to illustrate the application, benefits, and limitations of the method.

BIOGRAPHIES

Karl E. Wiegers is a software process engineer in a large product software division at Eastman Kodak Company in Rochester, New York. His 18-year Kodak career has included positions as a photographic research scientist, software developer, and software manager. Karl received a B.S. degree in chemistry from Boise State College, and M.S. and Ph.D. degrees in organic chemistry from the University of Illinois. He is a member of the IEEE Computer Society and the ACM. Karl is the author of the award-winning book *Creating a Software Engineering Culture* (Dorset House, 1996), as well as over 100 articles on many aspects of computing, chemistry, and military history. He is a frequent speaker at software conferences and professional society meetings.

Doris C. Sturzenberger is a software process engineer in a large division at Eastman Kodak Company. She has previously worked as a software quality engineer, tester, and developer on various product software and MIS projects. Doris holds bachelor's degrees in history (Indiana University) and data processing (Washington University), and master's degrees in library science (Indiana University) and history (College of William and Mary).

A Modular Software Process Mini-Assessment Method

Karl E. Wiegers and Doris C. Sturzenberger

Software process improvement (SPI) initiatives based on the Software Engineering Institute's Capability Maturity Model (SEI CMM) for software are often launched and tracked using comprehensive CMM-based process appraisals, such as the CBA IPI or the Software Capability Evaluation. However, such appraisals are too expensive and time consuming to be performed frequently. Several small-scale, incremental assessment techniques have been developed to take the process pulse of a software organization between full appraisals. Examples include a progress-assessment instrument employed at Motorola [1], the Interim Profile technique developed by the SEI with Pacific Bell [4], and a "spot check" approach used by the Norad System Support Facility [3].

CMM-based SPI initiatives are underway throughout the internal and product software development organizations at Eastman Kodak Company. Many of these organizations are using small-scale process mini-assessments to stimulate process improvement at the project level, to track progress toward higher maturity levels, and to assess an organization's readiness for a full-scale appraisal. This paper describes a flexible, modular mini-assessment method that permits construction of a customized activity sequence that best meets the needs of an individual project. This method has been successfully applied to many projects in several Kodak departments.

Background

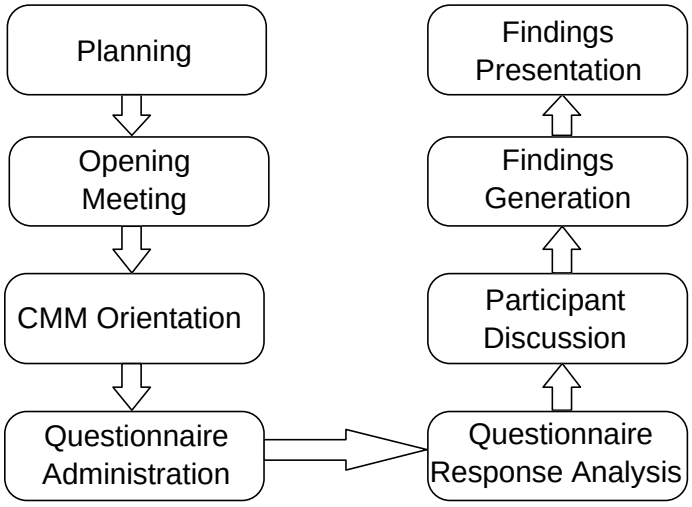
Over time, three distinct process mini-assessment approaches had been developed within Kodak. While their objectives were similar, they differed in the maturity questionnaire used, the sequence of steps involved, and the time commitment by both assessors and software project team participants. One method was essentially a two-day miniature version of a Software Process Assessment (SPA). Another variant required about 12 hours of contact time spread over several sessions, including an 8-hour session to conduct a practitioner discussion and generate findings by consensus. In a third, very compressed approach, one to three project representatives completed the questionnaire by consensus in a facilitated session. The assessors then generated findings by identifying performance gaps in all CMM key practices in the KPAs covered.

Members of the SPI community at Kodak wished to develop a common mini-assessment method that used a standard set of tools and procedures, yet accommodated the current methods so far as possible. The objective was to construct a mini-assessment architecture that could be tailored to each project's improvement objectives, life cycle status, team size, and time constraints. Using standard procedures and tools would make it easier to bring new assessors up to speed and would facilitate collaboration among assessors from different departments. The method developed must also yield consistent and reliable results. While the Modular Mini-Assessment method (MMA) described here has not yet been universally adopted across Kodak, it has been widely and successfully applied in several organizations.

Method Description

Figure 1 illustrates the overall process flow for the MMA. Several of these steps are commonly combined to reduce the number of meetings. The mini-assessment itself is separated from the follow-up action planning and action plan tracking activities. These essential steps may be facilitated by members of the software engineering process group (SEPG) supporting the assessed project's division, but they are ultimately the responsibility of the project itself.

Figure 1. Modular Mini-Assessment Process Flow.



The principal data gathering methods used in the MMA are responses to a process maturity questionnaire and an optional project participant discussion. All members of the assessed project are required to have one to four hours of CMM training to participate in the MMA. No managers above the project software leader level are involved in the data gathering activities. An explicit confidentiality agreement makes it clear that all data and mini-assessment findings are private to the project team, and that no data is attributed to individuals. The MMA is designed to use two assessors (lead and backup), although several steps can be performed effectively by a single assessor to reduce costs.

Although the MMA is based on the CMM, it was not specifically designed to comply with the CMM-based appraisal framework[2]. Consequently, the MMA cannot yield an official process maturity level rating. As the method is being used primarily to initiate and sustain SPI activities, we are more concerned with using the MMA to identify appropriate improvement opportunities than to generate maturity level ratings.

Planning

After a project decides to have a mini-assessment, the assessors work with the project leader (and perhaps the project's software quality leader) to plan the event. During a planning meeting, the assessors collect information about the project and the team members, and they describe the mini-assessment process to the project representatives. The assessors state their expectations of the project participants, and the project leader expresses his expectations for the mini-assessment experience. This planning meeting can also be used to educate the project leader further on the CMM and the SPI strategy supported by senior management for his organization, if necessary.

The assessors stress that the mini-assessment is only the first step on the path to improved software process capability. The real work consists of action planning and action plan implementation to address shortcomings in the current processes being used by the project. If the project leader balks at committing the time needed to follow

through on action plan implementation, we question whether the mini-assessment is worth performing at this time.

During the next step of the planning meeting, the project leader identifies the high-priority objectives for this mini-assessment. We present several typical mini-assessment objectives, and the project leader rates each of these as being of high or low priority ("medium" is not an option). Some of these objectives are: Identify process strengths and improvement opportunities, Serve as a catalyst for improvement, and Prepare for a formal CMM assessment. The pattern of high priority objectives helps the planning group select appropriate mini-assessment activities to make sure the objectives can be met. The project leader also decides whether the entire project team, or just a representative slice, will participate in the mini-assessment.

The remainder of the planning session is used to select the specific components that will comprise this mini-assessment, choosing from the options defined by the MMA method. Preliminary scheduling plans are made, and a representative from the project team is identified as a process liaison to be the prime contact between the assessors and the project team, and to assist with logistics. The deliverable from the planning stage is a mini-assessment agreement, which summarizes the objectives, participants, and events that were selected for this mini-assessment.

Component Options

The real flexibility of the MMA method comes from the multiple options available for each component step of the mini-assessment. The selections that are made affect the number of meetings held and the meeting durations. The time required for a mini-assessment ranges from 2 to 16 contact hours per project team participant, depending on the component options chosen. The options available for the MMA components shown in Fig. 1 are listed in Table 1 and described below.

Opening Meeting. The Opening Meeting is the kickoff event for a mini-assessment. It provides the first opportunity for the project team to hear what the mini-assessment is all about. The Opening Meeting can be held as a separate event (typically as part of a regularly scheduled project team meeting) or as a brief lead-in to the Questionnaire Administration session. As a separate event, more time is typically spent describing software process improvement and the organization's SPI strategy. In either setting, the Opening Meeting provides an excellent opportunity for the project leader and higher level managers to publicly state their support for the mini-assessment and the subsequent process improvement activities.

CMM Orientation. Four choices are available for presenting some background on the CMM and SPI to the project team members. A very short briefing (10 to 15 minutes) is always presented prior to administering the maturity questionnaire. This is usually enough refresher material for teams that have undergone a previous mini-assessment. Project teams having less CMM exposure can opt for a small briefing (about 30 minutes) or a large briefing (about one hour) presented. We strongly recommend that participants who are unfamiliar with the CMM take a four-hour in-house course on software process improvement using the CMM prior to beginning the MMA. The shorter briefings are presented by the assessors as part of the initial mini-assessment activities.

Questionnaire Administration. A maturity questionnaire is an important data gathering instrument for our mini-assessments. The project leader can choose which

questionnaire will be administered, as well as the participants who will supply responses.

Table 1. Mini-Assessment Component Options.

Component	Options
Opening Meeting	• separate event • brief lead-in to questionnaire session
CMM Orientation	• 10-15 minute refresher • 30-minute briefing • 1 hour briefing on SPI and the CMM • 4-hour course
Questionnaire Administration	1. Questionnaire selected • practices, subpractices, some institutionalization • all CMM key practices • institutionalization factors only 2. Administration mode • each participant completes a questionnaire • one set of consensus responses
Participant Discussion	• no discussion • discussion on selected KPAs • discussion on any process-related issues
Findings Generation	• assessors do off-line • participants do with assessor facilitation
Findings Presentation	• assessors present to project team • project team presents to their management

The basic questionnaire used was adapted from one developed by the Institute for Software Process Improvement (ISPI). It addresses many key practices and subpractices of the Activities Performed common feature of each KPA, along with some institutionalizing practices. ISPI also created a second questionnaire that addresses only institutionalization factors, and a composite questionnaire that encompasses all key practices of the CMM. Any of these questionnaires can be used in a mini-assessment, although we have the most experience with the first one. All of these questionnaires have possible responses that indicate the frequency of performance of a practice (Always, Usually, Sometimes, Rarely, Never, Don't Know, Not Applicable), rather than the Yes/No choices used in the SEI's maturity questionnaire [6]. Participants are also encouraged to write comments on the questionnaires. The KPAs covered by the questionnaire are selected during the planning session.

The second Questionnaire Administration option is whether each participant completes an individual questionnaire, or a single set of consensus responses is collected. The consensus approach is valuable for stimulating discussion among participants and clarifying their understanding, but it is impractical if more than a few project team members are involved. Individual responses (anonymous, of course) provide a broader cross-section of input from project team members.

Figure 2. Sample KPA Question Profile.

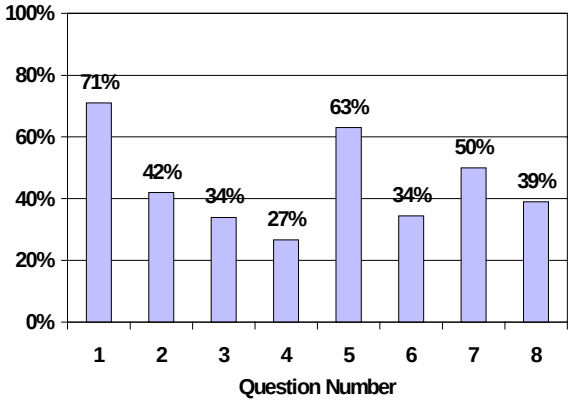
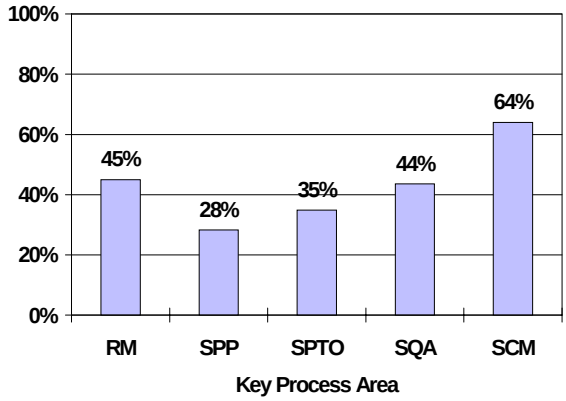


Figure 3. Sample Project KPA Profile.



The assessors facilitate the Questionnaire Administration session, using standard slides to describe the intent of each KPA before the participants answer the questions for that KPA. The group administration is necessary to help all participants have a common understanding of the questions. We only permit individuals to complete the questionnaire on their own under special, defined circumstances.

Questionnaire Response Analysis. The questionnaire responses are analyzed with the help of a Microsoft Excel spreadsheet tool that was created by ISPI and subsequently modified. The outputs from this tool are individual question response distributions, profiles of question ratings for each KPA (Fig. 2), and an overall project KPA profile that indicates a satisfaction percentage for each KPA (Fig. 3). To compute individual question ratings, the responses from individual questionnaires are weighted: An Always response receives a weighting of 1.0, Usually gets 0.75, Sometimes is worth 0.5, Rarely gets 0.2, and a response of Never has a weighting of zero. These weighted scores for all of the questions for a given KPA are averaged (all questions being weighted equally) to compute an overall percentage satisfaction rating for that KPA.

We do not attempt to issue a CMM maturity level rating on the basis of the questionnaire responses. In fact, we downplay the significance of a maturity level rating entirely, preferring to focus on the improvement opportunities revealed by the response patterns and by comments written by the participants.

The assessors study the questionnaire response profiles and supporting comments to compile a list of observations about the practice of each covered KPA by that project team. If a Participant Discussion is to be held as part of this mini-assessment, the observations constitute preliminary findings that are presented at the beginning of the discussion. However, if no discussion is planned, the observations are crafted into findings statements, identifying relative process strengths and weaknesses, consequences of the weaknesses, and recommendations for addressing the weaknesses.

Participant Discussion. In this supplemental data gathering activity, the assessors facilitate a discussion with the project team members, including the team leader unless he or she chooses not to attend. Though optional, we encourage holding a Participant Discussion if the project team can afford to invest the time. The additional information elicited by the discussion provides a more complete picture of the state of software practice in the project. The scope of the discussion (set during the planning

session) can be limited to CMM topics, or it can cover any process related issues that are important to the project team.

To open the discussion, the lead assessor presents the observations gathered from the Questionnaire Response Analysis. Then the project team selects up to three KPAs for further discussion. The process issues raised around these KPAs are used to generate mini-assessment findings.

Findings Generation. During planning, the project leader can elect to have the project team itself generate the findings with facilitation by the assessors, or he can ask that the assessors generate the findings off-line. In the first case, the Participant Discussion is extended by at least one hour and the assessors help the participants craft findings statements from the notes gathered during the discussion. Alternatively, the assessors develop findings statements after the Participant Discussion (or after Questionnaire Response Analysis, if no discussion was held), using all available data gathered from the project team.

In either case, the primary deliverables are up to three findings statements per explored KPA, along with actual or typical consequences of each finding, and recommendations about how to address each finding. We have compiled a database of all findings statements from completed mini-assessments to accelerate Findings Generation, since many projects face similar problems.

Findings Presentation. The last step of a mini-assessment is to present the findings summary to the appropriate audience. If the assessors generated the findings, they will present them to the project team. If the team generated the findings, they may choose to present the final findings slides to their own management. The scope of visibility of the findings is entirely up to the project.

Supporting Infrastructure

An extensive supporting infrastructure was developed to make the MMA repeatable, reliable, and efficient. Components of this infrastructure include detailed procedural guidance, presentation slide modules, forms, checklists, and a database that stores information about each MMA that has been conducted by the SEPG supporting the largest product software department at Kodak.

Procedural Guidance. The MMA procedural guidance consists of over 30 pages of detailed procedures for planning and conducting a mini-assessment. This document is updated periodically as we continue to gain experience and find ways to improve or extend the method. Each section in the guidance describes how to perform one of the component steps in an MMA. The entry and exit criteria for each step are stated, and checklists are included to assist with accurate planning and execution of that step. The roles and activities for the lead and backup assessors are also itemized. To assist with planning, tables are included to guide the selection of appropriate component options to satisfy each project's objectives for the MMA.

This document provides step-by-step procedure descriptions and implementation guidance to help the new (or rusty) assessor do a high-quality job. As we gain additional experience and learn what works and what does not, we collect that wisdom in this document. New assessors have stated that they found this detailed procedural guidance extremely helpful in getting them up to speed.

Slide Modules. We have developed more than a dozen Microsoft PowerPoint slide modules to be used for various presentation events associated with an MMA. These include:

- three CMM training modules of different lengths
- slides that describe the MMA process
- slides that are used during the planning meeting and other events
- a confidentiality statement
- slides that illustrate how questionnaire responses were analyzed
- templates for preparing observations and findings presentation slides

The ability to quickly assemble the slides to be used for each MMA activity from standard packets saves considerable time and reinvention for each mini-assessment. It also increases the repeatability of the MMA process. Slide module templates that can be quickly tailored for a specific project also save time and provide a consistent look to all of our MMA presentations.

Forms, Checklists, and Templates. The MMA infrastructure includes several kinds of tools to further streamline the execution of each mini-assessment. The project software leader receives a standard project profile questionnaire and a mini-assessment readiness survey prior to the planning meeting to collect background information. Forms were created to help plan a mini-assessment, record the time spent by each assessor on each stage of the mini-assessment, collect summary data, and obtain feedback on the experience from members of the project. An overall process checklist helps the assessors make sure that no task is inadvertently overlooked and that nothing is forgotten on the way to a mini-assessment meeting event. Templates are used to jump-start the processes of writing a mini-assessment agreement and creating a brief summary report that is submitted to the SEPG's management after each MMA is completed. As with the slide modules, these electronic aids save time and enable a repeatable MMA process.

Mini-Assessment Metrics Database. A database was created to store information about the mini-assessments conducted by one large Kodak department. The data stored include: the date, duration, and number of participants in each MMA meeting event; the questionnaire used, KPAs covered, and questionnaire results; the KPAs that were selected for process improvement activities; and the assessor time spent on each phase of the mini-assessment.

Having data available regarding the time spent on each step for multiple mini-assessments helps us plan our schedules and commitments more reliably, as well as letting us calculate the cost of each mini-assessment in total staff time. The data in this database let us track an organization's progress toward higher maturity levels, by aggregating results from multiple projects and seeing which KPAs are being pursued by various projects. Database queries can also generate reports to indicate how an organization is progressing toward its stated management goals related to SPI.

Experience Report

Table 2 summarizes some results obtained to date by departments that have applied the MMA method. The flexibility of the method results in a wide range of both assessor and project participant effort, depending on the component options selected. The assessor effort has begun to decrease as all assessors became fully trained and

experienced. The cost of a typical MMA mini-assessment is approximately $6,000, 5% to 10% of the cost of a SPA or CBA IPI.

Table 2. Results from MMAs Conducted To Date.

Number of mini-assessments conducted:	24
Average project team size:	12 (range 6-20)
Average contact hours/participant:	4 hours
Average assessor effort:	48 labor hours

The value of having a modular mini-assessment method is demonstrated by the fact that project leaders often choose different combinations of assessment components to create custom approaches that best suits their needs and schedule. Virtually all of the available component options have been selected by at least one project. Project leaders have expressed appreciation for the respect this flexible method shows for the realities and pressures their projects face.

Project team members also feel generally positively toward the mini-assessment experience. 79% of those who returned feedback forms (approximately 25% response rate) indicated that the amount of project time spent on the MMA activities was about right. 52% felt the mini-assessment would be very or somewhat helpful to their SPI efforts. 37% said it was too soon to tell, and only 11% indicated that the mini-assessment would be not helpful or detrimental.

The use of mini-assessments has generally been effective in introducing software project teams to software process improvement and the CMM, and in enabling those teams to begin focused SPI activities. Most projects have assembled working groups and begun to address the findings from the two or three KPAs investigated in the mini-assessment. A number of working groups have successfully executed their action plans, developing and rolling out improved processes in specific areas of software engineering. Other working groups struggled, experiencing little or no success. Our SEPG has developed standard working group materials and templates, and initial facilitation of working groups by SEPG members has contributed to several successes.

The strategy of using mini-assessments to launch and sustain SPI activities yields multiple benefits. A mini-assessment can be timed appropriately for each project team, while a full organizational assessment can be disruptive to projects having impending critical deadlines. Each project team can deal specifically with its own key process issues. The whole project team typically is involved, rather than just a subset of a large organization as in a SPA or CBA IPI. This facilitates the education of all project team members and leaders in SPI and the CMM.

By working closely with multiple software engineering teams through mini-assessments, the SEPG acquires a better understanding of the common challenges they face. The patterns of findings and known process shortcomings was used to direct evolution of the organization’s SPI strategy over time. The members of the SEPG gain credibility with the software developers through face-to-face, but non-confrontational, interactions during a mini-assessment. By working with multiple projects, the SEPG encounters opportunities to leverage solutions and improved processes from one project

to another. This minimizes the amount of invention each project must do on its path to improved processes.

This approach to process assessment is subject to the same risks that confront any SPI activity [5]. The most common point of failure we have observed is a lack of follow-through into action planning and action plan implementation following the mini-assessment. The timing of the mini-assessment is also important. If it is performed too late in the project life cycle, it can be difficult for the project to adjust its plans and schedule to permit time for action planning when deadlines loom. Also, later mini-assessments provide fewer benefits to reap on the current project, although they do position the team for greater success on their next project.

Some Kodak organizations have standardized on a specific pattern of component options for all of their mini-assessments. Those divisions that are more fully exploiting the flexibility provided by the method find that the ability to tailor a mini-assessment to best accommodate each project's situation is a significant advantage.

The MMA method does have some shortcomings. The absence of focused interviews and document reviews reduces the rigor of the mini-assessment, making it a less reliable predictor of the likely outcome of a CBA IPI. These components are being added to the method to provide additional data gathering and process verification rigor when required. In addition, senior managers are not as directly engaged in mini-assessments as they would be in a full organizational assessment. The SEPG has needed to take separate actions to get senior managers involved and keep them aware of our SPI activities, progress, and problems. Despite these limitations, the modular mini-assessment method has proven to be an effective component of a large-scale software process improvement initiative at Kodak.

Acknowledgment

The authors acknowledge the contributions made to development of the MMA method by Jeff Duell, Dave Rice, and Marsha Shopes, as well as the maturity questionnaire and supporting materials developed by Ron King and Linda Butler of Kodak and Jeff Perdue of ISPI.

References

1. Daskalantonakis, Michael K. "Achieving Higher SEI Levels," *IEEE Software,* vol. 11, no. 4 (1994), pp. 17-24.

2. Masters, S., and C. Bothwell. "CMM Appraisal Framework, version 1.0," CMU/SEI-95-TR-001, Software Engineering Institute, 1995.

3. Wakulczyk, Marek. "NSSF Spot Check: A Metric Toward CMM Level 2," *CrossTalk*, vol. 8, no. 7 (1995), pp. 23-24.

4. Whitney, Roselyn, Elise Nawrocki, Will Hayes, and Jane Siegel. "Interim Profile: Development and Trial of a Method to Rapidly Measure Software Engineering Maturity Status," CMU/SEI-94-TR-4, Software Engineering Institute, 1994.

5. Wiegers, Karl. "Software Process Improvement: 10 Traps to Avoid," *Software Development*, vol. 4, no. 5 (1996), pp.51-58.

6. Zubrow, David, William Hayes, Jane Siegel, and Dennis Goldenson. “Maturity Questionnaire.” CMU/SEI-94-SR-07, Software Engineering Institute, 1994.

Lessons Learned Implementing ISO 9001 in a Software Organization

Maureen Ganner
MedicaLogic
15400 NW Greenbrier Parkway Suite 400
Beaverton, Oregon 97006
(503) 531-7158
maureen_ganner@medicalogic.com

Mark Johnson
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, Oregon 97070
(503) 685-1321
mark_johnson@mentorg.com

Abstract

Effectively achieving ISO 9001 registration in a software development company can be difficuganizational context. This context includes a high rate of organization re-structuring, a high value on independence, and each group or individual determining for themselves if something will add value. This paper examines two different ISO 9001 implementation models used by Mentor Graphics Corporation in this software development context. The traditional cross-functional core team model is compared to a model of partnering with individual product development teams. The effectiveness of each model is presented, along with its strengths and weaknesses.

Key Words

Experience Report, ISO 9001, Software Process Improvement

Biographies

Mark Johnson is a senior software process engineer within the Infrastructure Team of the Strategic and Product Operations group at Mentor Graphics. The focus of his current work is guiding Mentor Graphics to ISO 9001 registration. He has previously worked on software engineering process improvement and software metrics. He has over 25 years of experience in data processing and hardware and software development. He has been active for the past 11 years with the Pacific Northwest Software Quality Conference as a presenter, organizer, and officer. He has written and presented papers on software process and quality topics at various conferences. He has written articles for a number of software publications. He presently serves as a special correspondent for Software QA magazine. He has earned bachelors and masters degrees in Computer Science.

Maureen Ganner is the Release Manager for MedicaLogic, Inc. In her current position, she is responsible for developing and implementing processes to increase the effectiveness and timeliness of software releases. She has previously worked as a software process engineer developing and implementing processes in the areas of project management and document control. She has also managed a customer support group and implemented numerous software systems as a project manager in the MRP II and public safety arenas. She has a bachelors degree in Business Management.

Copyright 1997, Mark Johnson and Maureen Ganner

Lessons Learned Implementing ISO 9001 in a Software Organization

1. Introduction

Achieving ISO 9001 certification for a software organization can be a daunting task. First, it can be difficult to interpret how the requirements of the ISO 9001 standard apply to a software organization. The standard's wording originated in manufacturing organizations and understanding how the requirements apply in a software organization is not always straightforward. Yet this problem seems minor compared to the difficulty of planning and implementing a process improvement program across a large software development organization whose culture values independent thought and autonomy.

Today’s software development organizations exist in a pressure cooker environment that requires new products and enhancements be brought to market in less time and at a lower price. Organizations must choose from a long list of priorities and choose the right ones or be overtaken by their competitors.

To survive in this environment, most software organizations are made up of teams of highly educated engineers, used to thinking independently and critically evaluating new ideas. To respond to rapidly changing markets, they frequently evolve and change their organizational structure. These and other factors mean that it is hard to get a software development organization to focus on and commit to a process improvement program like ISO 9001 registration.

This paper examines two different ISO 9001 implementation models used by Mentor Graphics Corporation to obtain ISO 9001 certification for its software development organizations. The paper starts with background information on the company and the ISO 9001 program. It then discusses how a traditional Core Team Model worked within a software development organization. Next, the impact of a major company re-structuring is described and how this caused the ISO 9001 Program Team to reconsider our implementation approach and move to a new model. The new model, called the Partnering Model, is described along with how it addressed the cultural issues of process improvement in a software development organization. Finally, our conclusions on the effectiveness of these two models are presented.

About Mentor Graphics Corporation

Mentor Graphics Corporation is a leading provider of electronic design automation (EDA) software for the telecommunications, automotive, consumer electronics, computer, semiconductor and aerospace industries. The company employs over 2,500 people in 10 sites around the world. The largest group of employees is based in Wilsonville, Oregon. There are also large development sites in San Jose, California and Warren, New Jersey.

Typical of many software development organizations, the culture at Mentor Graphics has a number of attributes that directly affect process improvement programs such as ISO 9001 certification. These attributes include:

- A strong sense of independence and autonomy, for both groups and individuals within the company.
- A strong focus on profit and loss, and remaining financially competitive at the product line or business unit level.
- Frequent organizational changes are made to maintain competitiveness as the markets for our products evolve.
- A staff of highly educated professionals with well developed critical thinking skills for evaluating changes in product technology.

These attributes make it a challenge to achieve the individual buy-in needed to make process changes within or across groups.

History of the ISO 9001 Program

Mentor Graphics' executive management first investigated the benefits of ISO 9001 certification in late 1993 and in mid-1994 assigned the first resources to the ISO 9001 implementation effort. While other process improvement models were investigated, the ISO 9001 standard was selected to provide a baseline foundation from which further process improvement efforts could leverage. The objective of ISO 9001 registration was to formalize existing processes while minimizing changes to these processes or the need to implement new processes.

The initial goal of the program was to achieve ISO 9001 certification for all North American product development, customer support, and administrative and manufacturing sites in approximately an eighteen month period. The effort would encompass over 1,000 employees in eleven different divisions. An ISO 9001 Program Team was formed to lead the effort.

2. The Core Team Model at Mentor Graphics

The structure of the Core Team Model was to bring together one representative from each functional area of the company participating in the ISO 9001 registration effort. This model of a cross-functional team to complete a special project is a text book approach and was recommended by the consultants who were advising our ISO 9001 efforts. This approach had proven successful for their other clients, especially in tightly controlled environments, such as manufacturing organizations.

The Core Team Model at Mentor Graphics actually consisted of three levels of teams. At the top was the ISO 9001 Program Team which consisted of the program director and two ISO process 'experts.' The program director had overall responsibility for driving ISO 9001 registration, and the process experts acted as staff to help plan and manage the program, along with consulting internally to the company on ISO 9001.

The next level was the Core Team. This team was made up of one representative from each of the North American product development divisions and representatives from customer support, the legal department, human resources, information resources, manufacturing, purchasing, distribution, plus the ISO 9001 Program Team. The representative from each organization was expected to become an ISO 9001 expert for their organization and act as a communication and decision-making link between the Core Team and their organization. The Core Team would discuss common issues facing the ISO 9001 effort and make implementation decisions. The Core Team also formed working sub-groups to develop specific items needed for implementation. For example, a sub-group was formed to develop a set of documentation templates, procedures and a document control standard to support the documentation of policies, procedures, and standards needed by the company.

The third level of teams were implementation teams. Each organization was responsible for creating an implementation team to carry out the details of completing the actual ISO 9001 implementation work within their organization. The implementation teams were led by their organization's Core Team representative. For example, the Customer Support implementation team defined the full life cycle of processes involved in preparing for and supporting customers. They then refined and documented each of the needed processes.

Core Team Model and the Organizational Context

The key cultural attributes of Mentor Graphics, mentioned in the section About Mentor Graphics Corporation, had a strong impact on the effectiveness of the Core Team Model. Following is a summary of the key factors affecting the ISO 9001 registration effort under the Core Team Model.

Factor 1: The strong sense of autonomy and independence of the different software development organizations within Mentor Graphics was in direct conflict with the idea that one team, the Core Team, would make common implementation decisions for the entire company.

As an example, the Core Team made the decision that an on-line documentation system, built using the emerging 'web' technology of HTML browsers accessing hyperlinked structures of documents on the corporate intranet, was the best way to meet ISO 9001's document control requirements. However, web technology was at its early phases and one division general manager would not support the investment of developing a web-based implementation. So, in that organization, effort towards the on-line implementation was not started.

Factor 2: Mentor Graphics has a profit and loss business model down to the product line level. This business model places a strong emphasis on revenue generating projects and evaluating activities to determine if they will add value for your group. Since there was no direct link between ISO 9001 registration and increased revenue or financial value for individual organizations, supporting the corporate goal of ISO registration was not a high priority for most product development groups.

Factor 3: Frequent restructuring in the development divisions led to a high rate of turnover in Core Team membership. For example, if a division was split into two groups, a new Core Team representative would have to be added for the new group. Or with the reshuffle of groups within a division, the primary job of a Core Team representative might change and not leave them any time to spend on Core Team work, requiring a replacement. In either case, each time membership of the Core Team changed implementation efforts slowed while the new member was brought up to speed. A new member would have to be trained on the ISO 9001 standard, be briefed on the implementation plan, what had been completed, and what was still outstanding.

Factor 4: Software engineering staff members are highly trained and are used to evaluating new ideas and deciding independently if these ideas make sense for their projects. They brought this same mindset to how the ISO 9001 requirements should be met. Since decisions made by the Core Team were by consensus, as membership changed, new members would need to be brought into concurrence with previously made decisions, or previously made decisions would have to be modified to accommodate their views. It was not uncommon that decisions and work already completed would be re-examined and re-implemented to satisfy the requirements of new team members.

All of these factors combined to make the Core Team Model ineffective in the software development environment. Decisions were slow to be made, delaying progress. As implementation stretched out, representatives to the Core Team would change, causing further delay. Organizations that were not truly bought into the idea of ISO 9001 registration and did not see any value for themselves achieving ISO 9001 certification, did little work within their implementation teams. The only groups to make progress on their implementation goals were non-software development groups, such as Customer Support and the legal department.

Benefits of the Core Team Model:

- Provided a potential for cost savings by developing common methods and processes for use across all organizations.
- Allowed for the formation of cross-organizational working groups to develop common process.

Deficiencies of the Core Team Model:

- The individual software development organizations resisted deploying common approaches developed by a central body (For example, the Core Team or one of its sub-groups).
- The ISO 9001 registration effort was not tied to the profit and loss goals of individual organizations, rather it was a corporate objective.
- The turnover in Core Team membership slowed the team down by having to bring new members up-to-speed and by causing the re-evaluation of past decisions.
- The need to sell individuals on each Core Team decision made the decision-making model consensus. These consensus decisions were difficult to maintain, and hard to get carried out by the various implementation teams.

3. A Major Re-Structuring

In the forth quarter of 1995, a major re-structuring occurred within the company. A key feature of this re-structuring was even greater decentralization of decision-making with the creation of Strategic Business Units (SBUs). These SBUs were given a high degree of autonomy and the responsibility to determine how best to organize and define processes to meet their business objectives, within a thin layer of high-level corporate constraints.

Facing a major restart of the ISO 9001 registration program after this re-structuring, the ISO 9001 Program Team evaluated the slow progress to date and determined to make two major changes to the program. First, the scope of the initial registration effort was re-examined, and a set of pilot product development teams were nominated by their management to participate in the initial registration effort. As part of restarting the program, a gap audit was performed for each nominated team to help them identify the areas where they would need to focus their efforts. The non-software organizations, such as Customer Support and the legal department continued their implementation work as before.

Second, it was decided to change the organization model of the ISO 9001 registration effort from the Core Team Model to a Partnering Model (described in the next section). Reducing the number of software development teams participating in the registration effort permitted the change to this model. While the Partnering Model required more ISO Program Team resources, it was decided this was an acceptable tradeoff for overcoming the ineffectiveness of the consensus decision process and the lack of accountability and support for implementation within the product development divisions.

4. The Partnering Model at Mentor Graphics

The foundation of the Partnering Model is to provide an 'ISO expert' from the ISO Program Team to work directly with (partner with) each product development team and functional area team participating in the ISO 9001 registration effort. This is a part-time assignment, with each ISO expert supporting several teams. In addition to supporting teams, the ISO Program Team members took on responsibility for developing required infrastructure pieces, such as an executive-level management review process, corrective action process, and an internal audit process.

In their partnering role, the ISO expert is responsible for helping each of their partner teams: understand the requirements of the ISO 9001 standard; consider alternative approaches to implementing processes; provide training to their partner team members; monitor the progress of their partner team’s implementation efforts; and identify additional resources if registration goals are at risk.

Each project or functional area team identified one member to be their ISO Implementation Leader. This is generally the team leader, but in some cases is a member of the team with a special interest in process improvement or the time available to work on process improvement efforts. This person is responsible for leading

the implementation effort within their team. This responsibility includes: allocation of implementation tasks to team resources; managing the development of processes and process documentation; managing implementation schedules; and participating in the internal audit program.

In addition to one-on-one meetings between the ISO Program Team partners and their product or functional team counterparts, biweekly Partners meetings are held. All of the product team and functional area ISO Implementation Leaders and the ISO Program Team attend these meetings. The meetings are a forum for sharing details on the interpretation of the ISO 9001 standard, sharing implementation problems and solutions from various teams, and reviewing general issues which could affect the overall ISO 9001 registration effort.

Partnering Model and the Organizational Context

The Partnering Model was implemented to overcome the problems the Core Team Model encountered with the culture of the organization. Following is a summary of the key factors affecting the ISO 9001 registration effort under the Partnering Model.

Factor 1: The strong sense of autonomy and independence of the software development organizations was supported by the Partnering Model. The ISO Program Team partner helped each product team identify where they already had processes in place that met the requirements of the ISO 9001 standard. Working closely with the product or functional team lead, they were able to gain buy-in for the few company guidelines that all teams are expected to follow. The ISO expert then helped the teams define and document new processes needed to be ISO 9001 compliant. The goal was to complete this work with a minimal impact on the team's resources, as well as having a minimal impact on how the team conducted its day-to-day work.

Teams were encouraged to challenge the traditional methods for documenting processes and to document things in a way that met their individual team needs. For example, instead of creating a new process document describing the process for making design changes, a team could add a section to a project plan to describe the process they are using for that project. Another example is creating a web page that lists the specific skills a new engineer on the team would need to learn, and how they could develop those skills (on-the-job training, classes, reading) to satisfy the ISO 9001 requirement to identify employee training needs.

Factor 2: The Partnering Model fit with the profit and loss business model by making sure that work done by each project team for ISO 9001 implementation added value for them. Achieving ISO 9001 registration was part of the compensation plan for management of the groups piloting ISO 9001, as it had been under the Core Team Model. With the closer relationship of the ISO Program Team members to the specific projects, clear tracking of progress could be reported on a monthly basis, and project team management made sure that ISO 9001 registration remained a priority.

Factor 3: By working with groups at the product development team level, instead of the division or business unit level, the Partnering Model overcame the problems of frequent

organizational changes. Product development teams generally remain intact through these organizational changes and can remain focused on ISO 9001 implementation work even when restructuring occurs. To maintain management commitment after a restructuring the ISO Program Director meets with the new division or business unit management to confirm the organization’s continuing commitment to the ISO 9001 registration effort.

Factor 4: The Partnering Model overcame the skepticism and critical review by each individual that was so problematic for the Core Team Model. Although consensus is hard to reach across large organizations, generally teams of individuals who are working closely together develop 'norms' for how they will communicate and perform their work. These norms could be built upon in identifying processes which met ISO 9001 requirements. In fact, all of the project teams achieving ISO 9001 registration reported that it actually helped them by clarifying processes and communications they had always assumed were commonly understood, but were found during internal audits to differ from person to person. Also, the cross-organizational meetings held biweekly became forums for sharing problems and solutions. They were no longer viewed as forcing unpopular processes on teams, but rather a chance to air concerns and build on good ideas from other groups.

These factors combined to help the Partnering Model build on the cultural attributes that had been the downfall of the Core Team Model. Although processes were identified and documented in ways that made sense for each product team, there is duplication of effort because there are several documented processes for the same ISO 9001 requirement. For example, each project team has their own method defined for planning and completing design reviews.

Benefits of the Partnering Model:

- The ability to define processes at the product team level allows for the autonomy and independence of software development teams.
- Working closely with each team in identifying processes ensures that the team feels the resultant processes add value to their efforts.
- Working at the product development team level reduces the impact of organizational changes since teams generally remain intact through these changes.
- Taking a team-level value-added approach to fulfilling ISO 9001 requirements provided buy-in from individual team members.
- Meeting regularly to discuss similar problems created synergy in sharing solutions to issues.

Deficiencies of the Partnering Model:

- This model is labor intensive for the ISO Program Team. Only a limited number of product or functional area teams can be partnered with at one time.
- Working with teams one-on-one with a focus on what adds value for an individual team means that multiple solutions may be developed for similar problems.

5. Conclusions

Effectively achieving ISO 9001 registration in a software development company can be difficult because of the cultural attributes. These cultural attributes include a high value on independence and autonomy for groups, a profit and loss business model in a highly competitive market that focuses groups on revenue generation, a high rate of organization re-structuring to meet the ever-changing market, and the training of individuals to critically evaluate any changes to determine if the change will add value for their specific work.

At Mentor Graphics, we found that the traditional Core Team Model for structuring an ISO 9001 registration program did not work well in the software development groups. An evaluation of the Core Team Model led us to the conclusion that it was in conflict with the cultural attributes of a typical software development company.

To achieve ISO 9001 registration, we developed the Partnering Model described in this paper. We found this model to be effective in addressing the cultural attributes that had been problematic for the Core Team Model. In addition, our implementation of ISO 9001 in the software development teams has been considered helpful in clarifying common processes and methods, and ensuring that necessary communication is occurring within the team. The Partnering Model is more labor intensive for the supporting ISO Program Team and can lead to multiple approaches to meeting the requirements of ISO 9001. In spite of these additional costs, we would recommend the Partnering Model to any software development company with a culture similar to ours.

Facilitating Change in the Software World

James R. Bindas

Intel Corporation
JFT-101, 2111 NE 25th Avenue
Hillsboro, OR 97124
Phone: (503) 264-8869
jbindas@ichips.intel.com

Abstract

What exactly is change in the software world? Change is a commonly used term but has many different meanings. This paper explores the perspective that change is actually an equation that includes the factors of reinforcement, reaction, state, culture, time and the element of change itself. The paper explains each factor and its importance in the equation of change.

Keywords

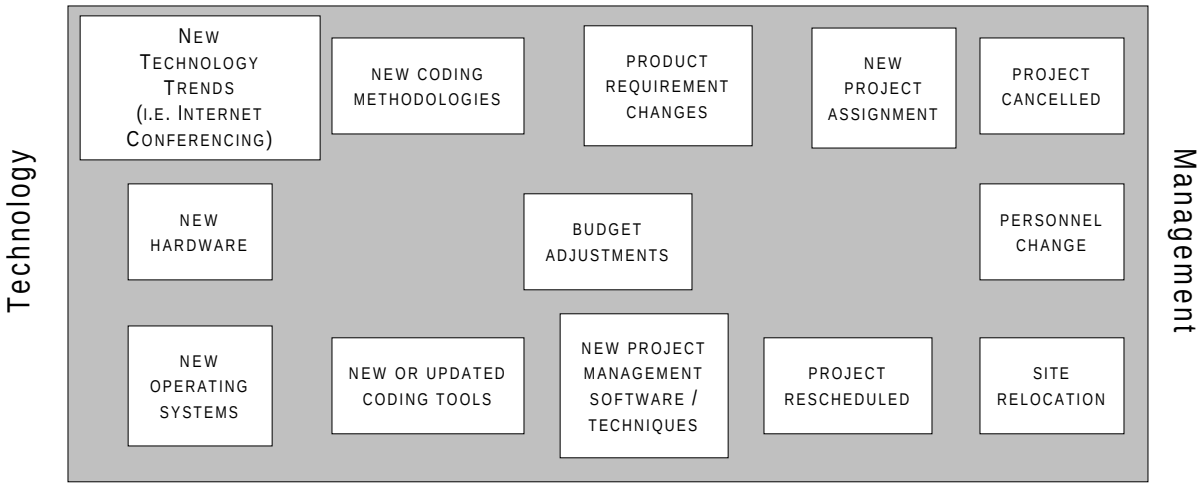
Change, software, reinforcement, resistance, reaction, state, and culture.

Biography

James R. Bindas is a Software Process Engineer with Intel Corporation in Hillsboro, Oregon. His tasks include working with the software development community to help standardize and improve software development processes. James has been working with Intel for eight years in various software roles ranging from software tester to project leader. Before joining Intel James worked for RCA/GE Solid State and Harris Semiconductor as a Quality Assurance Technician. James holds a Master of Science degree in Computer Science from Steven's Institute of Technology, and a Bachelor of Science in Graphic Communications from California University of Pennsylvania.

Introduction

What exactly is change? In the software world, which introduces and retires paradigms on a regular basis, the word change evokes sharp, distinct, and diametrically reactions. Although few people would say that change is not necessary, reaction to change is often extreme. Change and reaction to change are fascinating, partly because change does not always bring about the intended effects. This paper includes accumulated knowledge, experiences, and observations about making a successful change. In the software industry, many elements are introduced at a frantic pace which can be overwhelming. When we try to analyze different types of change, they seem to fall between the spectrum of technology and management, as illustrated below:



On further examination, many changes are simply “unavoidable,” and are an integral part of the work environment. For example, a key manager or engineer leaves the project in a state of chaos, budget cuts prevent the purchase of updated equipment, software revisions or new standards leave a product outdated. This can lead to a number of reactions ranging from frustration to outright rebellion.

With this in mind, I choose to think of managing change not as introducing and integrating change, but as controlling the change environment. Change will happen, whether it is introduced internally or thrust upon you by outside agencies. How we manage change is just a question of our position and reaction when it occurs and what it will affect.

While many changes in the software world are unavoidable, many can be controlled. Introducing change into our environment looks like a much easier task to perform than it may be. Many times, I will refer to a fictional software process engineer named Rob. Rob represents a composite of my experiences and the experiences of fellow software process engineers.

My initial understanding of change could be described by the simplistic model below. At first, I believed that by introducing a change element, the desired change in behavior would occur. As my example outlines, this is not always the case.

Change = Change Element

Rob was working as a test lead. His group ran a series of test scripts on their platforms. He wanted to track the results electronically, by placing them in a central database. Using this method, his manager could track the results from her desk. Some of the testers followed instructions; most did not. His attempt to institute a change was unsuccessful.

Looking back, one of the reasons Rob failed in getting people to log their results was because logging

results was not valued. Rather, they saw the change as time spent performing busy work. Management also did not see the value of the change, and rarely checked the database for results, reinforcing the attitude of the developers. The lesson in this example is that understanding the culture is key in understanding which changes will be supported and which will not be supported. Therefore, my change model was adapted to include corporate culture.

*Change = **Culture** + Change Element*

Understanding the Culture

Neal Brenner[1], a veteran in making change, believes in studying and understanding the culture of the organization in order to work with it.

He feels the authoritarian approach only works in authoritarian cultures. An example where the authoritarian approach might work is in an organization that does business with the Department of Defense (DOD). Organizations that supply software to DOD are contractually mandated to achieve certain levels of the Capability Maturity Model (CMM). Employees understand the consequences of not scoring well in the assessment/evaluation. One disadvantage of using an authoritarian approach is that people tend to focus on how to pass the CMM evaluation versus implementing real software improvement.

In an enterprising organization that operates on data to make decisions, no change will be considered or implemented unless data is available to show benefit in the form of improved productivity or efficiency. Most importantly, the change and its benefits must be presented in an understandable fashion. For example, Rob was given an assignment to introduce a documentation review process to a department. The review process was sound, but the department hesitated to consider changing it. The department needed to make its numbers and introducing an unknown process was risky. They would rather stick to their own processes, with known results. But, once Rob presented metrics from other projects that showed positive gain, they broke the "not invented here" paradigm. Did the metrics alone convince the department? No, but now the department was ready to consider this new process.

Some enterprising organizations don't operate exclusively on data. Instead, its people operate on principles. While presenting data can assist these organizations, they examine whether the improvement you are proposing matches their model of the company.

For some organizations, enacting change boils down to terminology. Depending on the organization's history, certain terminology will unleash strong feelings. For example, Rob found the word "process" received much less resistance when substituted with the word "methodology." The substitution is not an exact match, but it worked.

One of the other big benefits of understanding the culture is an understanding of the different states that exist, and the correct time to introduce change.

*Change = **State** + Culture + Change Element*

Many industries run in cycles. The state the organization is in can determine the type of resistance one will get in introducing change. Identifying when to introduce the change is a hard element to master. The first step is to identify the different states of change. Norm Kerth's workshop*, Leadership from the Technical Side of the Ladder*[2]*,* claims that there are four different states of change: *Status Quo, Chaos, Integration and Practice, and the New Status Quo.*

In the "Status Quo" stage, people are very familiar with their jobs. The job functions themselves are

---

[1] Bindas, James. R. "Meeting Minutes between Neal Brenner and Jim Bindas". (Nov. 8, 1996).
[2] Kerth, Norman, "Leadership from the Technical Side", Pacific Northwest Software Quality Conference Workshop, (Oct. 19, 1996), p. 43.

routine, and provide a small amount of improvement to the overall process. Management adopts a philosophy of low risk, high predictability on results. I would like to call this the “Don’t rock the boat” state. Rob’s former manager once told him that he was always ahead of his time by introducing change before it could be accepted. The question remains, how do you get people to make change if they are always in the status quo state?

One method is to stop implementing the change and wait for an opportunity, such as a crisis or a "Chaos" state, to present itself. It is usually triggered by a number of things: switching operating systems, losing a contract, slipping product quality, or just a “do or die” situation . During this state, people are more open to change, in order to bring the crisis to a resolution and relieve the pain. An example of this was when Rob had an idea for a feature enhancement for our product. Instead of telling people about it, he waited until people were complaining that the product needed this feature. He promptly raised his hand and explained that we could do it with our present hardware. Sure enough Rob project was funded, and the feature was implemented within the month.

Unfortunately, this technique depends on a crisis to erupt. This leaves us on hold in the state of status quo never having the opportunity to present our ideas. According to James A. Belasco’s book, *Teaching the Elephant to Dance: The Manager’s Guide to Empowerment,* we often fail in making changes because people fail to prepare for the change before introducing it. He claims that one of the first steps in making change is to create a sense of urgency to move from the status quo state to a crisis or chaos state. Many times, products are introduced to change the industry, only to fail. The public was not prepared for the need of the revolutionary product. Rob has friends who are true believers in IBM's OS/2 operating system. They claimed that it was years ahead of Microsoft Windows. But, very few people are buying it. Why? No one really stated a good case on for moving to the new operating system.

Looking back on Rob’s attempts to bring about change, the people who were the targeted did not see the need for change. Rob was in the chaos state and ready for change, but the targeted group was still in the status quo state. Rob was able to remedy this mismatch by arguing with facts and presenting the case to people who are the most sensitive. This lesson is a must for software testing. Developers are often overloaded in correcting defects. They focus on correcting the defects that have the most benefit. Therefore, it is up to the tester when filing the defect to not only state the defect and how to reproduce it, but state the consequences of NOT fixing the defect.

Another example of this approach concerns a product release process. Rob was put in charge of a product release process. While the end product was of high quality, he found that the process was not efficient. Instead of trying to change the process, he tested the current process and exposed the issues. Once it was determined that his issues were real and they impacted the delivery time to customers, management eagerly listened to his suggestions and drove the improvement changes accordingly. After some initial time investment setting up the process, the actual product release time decreased, while the quality of the product was not compromised. The key lesson is that Rob did not drive the changes. He exposed the issues and brought management into the chaos state. Once in the chaos state, management was eager to resolve the issue and drove his process suggestions.

Once change is introduced and accepted, it signals the beginning of the “Integration and Practice” stage. Arguments are no longer on whether the change will be agreed to, but how it will be executed. However, if a crisis occurs at this state, it is common to transition to the Chaos state.

Finally, change is implemented and institutized. This marks the final state of the “New Status Quo,” as well the last state of change.

As this section mentions, the best time to introduce change is during the chaos state, but it is the most risky state as well. Because, the reaction to the change can lead change away from the intended target.

*Change = **Reaction** + State + Culture + Change Element*

Rob learned when playing the game of billiards that it does not always matter if he sinks the desired ball,

but where the cue ball lands after the shot. If the cue ball is not set up for the next shot, the victory of sinking a single ball is short-lived.

Making changes is similar. When change is introduced, people will react, sometimes positively, sometimes negatively. If the negative reaction is not recognized and addressed, resistance will set in and change is either never adopted, poorly implemented, or it can stop a project instantly.

Is what we perceive as resistance actually resistance? Dale Emery does not think so.[3] Emery believes that resistance does not exist; instead we receive a response. This response is filled with information that is vital to the person making the change. Many people will ignore the information contained in the response and call it resistance, while it can be valuable and should be listened to. Is it valid? The validity of the response does not even matter. Unless a person's opinion is understood and addressed, they will not become a part of the change process.

To illustrate this point, Emery suggests this exercise:

1. Go up to another person, and stand facing him/her. Ask them to participate in an experiment with you.
2. Place your hands up, palms out towards the other person. Ask the other person to do the same so that your palms are touching theirs.
3. Say, "I'm going to push gently on your hands." Then push gently and observe the reaction.
4. If the other person pushes back, question their reason for doing that. Notice the first thing said.
5. Do this to an additional ten or so people.

When Emery performs this experiment in his group workshops, normally people push back for the following reasons:

1. "I don't know."
2. "I pushed back because he pushed me."
3. "I thought that was what you wanted me to do."

Would you call this "pushing back" resistance or a response?

The key point is that, when facilitating change, do not forget the people who are enacting the change. Their input is needed throughout the process, and they need to feel that they are tied to the process. Otherwise, the process will turn one way and people's inertia may take them another. This can appear to be resistance, when in fact they are telling you that they are separated from the process.

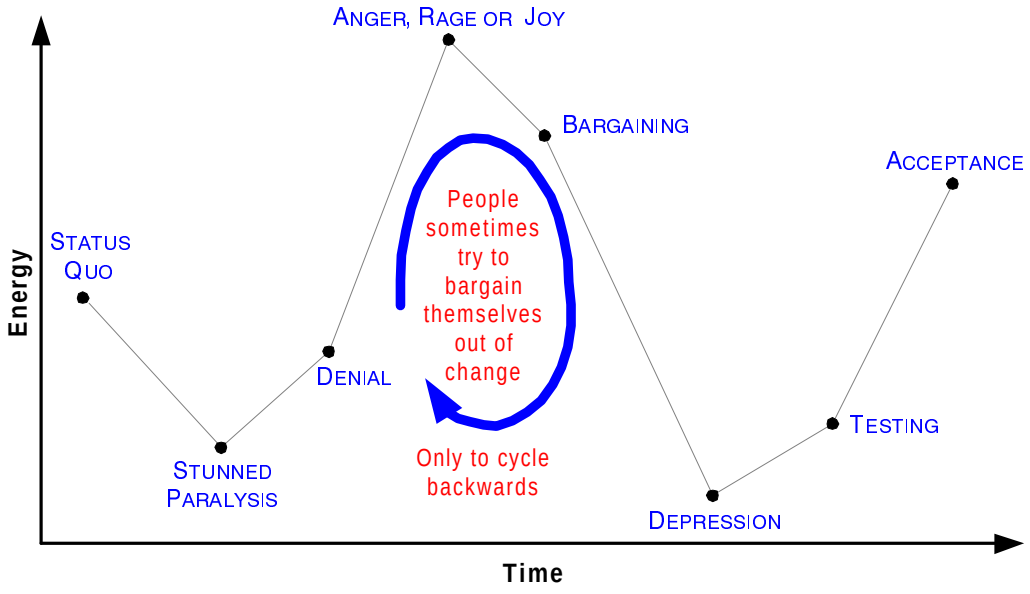
An example supporting this theory occurred when Rob was a program manager of a process improvement group. Rob was meeting with his working group when a member started to resist the direction being taken by questioning the value of everything being done. Rob came away somewhat frustrated, because he believed team member had already accepted what we were doing. Rob met separately with the team member later. She confided in Rob that she was frustrated on the lack of progress and was questioning everything in an attempt to get the group moving again. She was amused when Rob mentioned that he was frustrated as well. They aired all of their frustrations with each other. They understood each other, and were able to finish the outstanding issues with remarkable ease.

Some people see resistance as good. Rob actually seeks out resistors. He pulls them aside and speaks to them on a personal level about the change. He ends up using them as a sounding board. He told me that, if he is able to get acceptance at a personal level, when he brings the change to the greater working group there is a minimum of discussion and the change is adopted. He calls this process, "pre-selling the idea."

---

[3] Bindas, James. R. "Meeting Minutes between Dale Emery and Jim Bindas". (Nov. 9, 1996).

Dealing with change is difficult. It is not uncommon for people to have a difficult time adjusting to change. A few years ago, Rob job status changed. The change was not welcome. At first he was stunned and refused to believe it. When he realized that it was real, Rob became upset. He thought there was no way this could be happening to him because they made a mistake. Finally, he accepted the status, tested the waters, and acknowledged the situation. At the time, he was going through the seven standard responses to change as defined by Elizabeth Kutler-Ross in her work, "On Death and Dying."[4]



Response to Change [2]

Her model shows that all change evokes these responses. Depending on the type of change, navigating through these stages can be easy or difficult. An unfavorable change may cause you to be stalled in one of the stages. For example, if a project is being canceled, employees may stay in the anger stage for a time and blame their supervisors. On the other hand, when change is favorable, it is a lot easier to make it through all the stages. An example can be getting the job that you are seeking.

The first stage is **Stunned Paralysis,** or the "it is not happening to me" stage. This is a very common reaction to any change, good or bad. An example is when it was announced that my site was being closed. Rob had heard rumors of it for a few weeks, but actually hearing it being announced was shocking. Rob thought to myself, "Is this actually happening?"

The news was announced off-site. Rob did not know if Rob was being transferred or being let go. They told the site that most people would be transferred to Arizona, where he did not wish to go. He kept telling himself that he didn't want to go to Arizona. This was his **Denial** stage. Rob refused to believe that he was being transferred.

Later in the day, Rob was offered a job in Oregon, where both his wife and I wanted to move. They were both excited to hear the news. At this point, Rob entered what I called the **Joy** stage. However, if the offer was to Arizona or no offer at all, Rob energy levels would still be high, but he would have been filled with any where from **Anger** to **Rage**.

Neither he nor his wife had ever visited Oregon. Upon their first visit, ehy enjoyed the state's beauty, but the housing situation was not what they expected. Rob and his wife thought they would be able to afford their dream house but it was not affordable. This is the **Bargaining** stage. Reality threatened to set in on

[4] Kubler-Ross, Elisabeth, On Death & Dying. (Hudson River Editions Ser:Macmillan, 1991.).

them.

Rob and his wife realized their dream house would remain a dream. They debated moving. Perhaps, their situation would be better if they stayed in their present location. Rob and his wife found themselves at the **Depression** stage. Rob returned to the **Bargaining** stage when he began interviewing for other jobs outside his company. Moving back to previous stages is quite common and one can become cyclical. Debating the move put him in a cyclical state, draining valuable energy from him while accomplishing nothing.

After re-examining the housing section in the newspaper and visiting Oregon for a second time, Rob and his wife agreed that they could still buy a much nicer home in Oregon than the one currently owned. Perhaps, moving wasn't a bad idea. This is the **Testing** stage, where they began to trade off values and features to see if accepting the change is feasible.

Note the difference between the **Bargaining** stage and the **Testing** stage. In the **Bargaining** stage, Rob and his wife thought their dream home was attainable and refused think it was not. They kept blaming external factors that were preventing them from attaining their dream home. In the **Testing** stage, they realized that their dream home was NOT attainable and began to trade off features from their dream home for a home they could afford.

Rob and his wife moved into the **Acceptance** stage by buying a nice house in a good neighborhood. They realized that they made the right decision. Do they still wish for their dream house? Certainly, but they acknowledge their situation and hope that one day they will own their dream home. Until then, Rob and his wife are enjoying what they have.

Getting through these seven steps is accepting change. However, it may be pointless, unless there is a mechanism in place to support the change over time.

*Change = **Reinforcement** + Reaction + State + Culture + Change Element*

Rob was assigned a project to centralize all of the software processes into a single repository. When Rob's portion of the project was completed, it successfully achieved the first goal of creating the repository. However, Rob noted that although tremendous effort was put into getting the project off the ground and presented to the department, no one put in any reinforcement or feedback mechanisms to support the project long term. Also, none of the senior management staff issued any directives to support the project. It was implied that Rob would be able to carry on the project afterwards by collecting the processes. As it turned out, after the initial successful launch the project lost importance. When Rob attempted to contact lower-level management for assistance with gathering processes, none assisted. Why? The perception was senior management did not support it by contributing processes to the repository, which effectively ended the project.

If an improvement project is to succeed, top down directives need to be issued to the rest of the organization that this is a supported project. Otherwise, the effort will be a grass roots effort only. While there are many successes in grass root campaigns, the likelihood the campaign will succeed is small. For a grass roots campaign to grow and prosper, it needs to draw upon resources to reinforce the changes made. If management controls the resources, its growth will be cut short.

Perceived benefit is often overlooked as a reinforcement agent. People often look upon the advantages and never look upon the impact that change brings to the intended targets of the change. A year later, the same department was beginning a new product line. Since the current product line had their processes established, it was only logical that they should work from proven processes. Senior management agreed and a directive went out to copy the old product processes. The problem was none of the processes were detailed in the same location. This created a need to re-create the software process repository. As for Rob, who could not get any support, he is now leading the current project and being supported by every management level needed.

Tying perceived benefits to the change acts like a reinforcement tool to support the change long term. Perceived benefits vary. Neal Brenner likes to use the term "remove the pain[5]". He feels that removing the pain is the most powerful tool for the change agent as well as being the most common response to facilitate change. Two approaches used for relieving pain are:

1. Understand the pain and remove it
2. Remind them of the pain that will be encountered unless procedures are put in place to prevent it. Many times organizations will skip part of the release process, in order to rush the product to market.

Many times having hard data will bring perceived benefit to management by showing trends, or the history of similar efforts. For example, "are we on track for product rollout?" or, "if the past effort took x hours, do we have x hours in our schedule to implement this task?" The reality of the current situation can be established by comparing it with past experiences. Hard data will help determine the organization's focus.

Metrics themselves can present problems. Once, when Rob presented metrics to management, they questioned the applicability of the data to the current situation. This resulted in Rob collecting new metrics. The process was frustrating, and I am not sure if he was able to present the new data to management or was forced to abandon this approach. Another example involved being in a rush for instituting metrics for a department. They were collecting data on every possible type of situation. Results were impressive but meaningless, resulting in low credibility.

Metrics can provide a powerful reinforcement to change. However, they need to be used and presented carefully to establish credibility. The answers received from questioning are only as good as the questions and the data itself. Vague questions lead to vague answers. Specificity in the questions will obtain meaningful information that will bring the organization into focus.

Even when everything has been done right up to this point, they are no guarantees that the change will take. Sometimes, it just takes time.

*Change = **Time** + Reinforcement + Reaction + State + Culture + Change Element*

Time is the most underestimated part of making change. I remember a country music singer saying it took a lot of work and time to become an overnight star. As in the previous software repository examples, first the project appeared to fail, but a year later it was more alive than ever. Time is the measurement of how long it took the change to take place. During this period, the change agent must perform a number of tasks to maintain the momentum. That is why a successful change agent needs the following qualities:

- Vision
- Persistence
- Confidence
- Optimism

These are the values that Norm Kerth[6] speaks of in his workshops. Norm breaks down these values accordingly:

Vision

The word "vision" can be defined in a dictionary, but the concept itself is an abstract perception of a possible reality. Vision is only seen in one' s own mind. Having a vision can set the direction for the project. Vision can see beyond the obstacles that block the path, into a better situation.

---

[5] Bindas, James. R. "Meeting Minutes between Neal Brenner and Jim Bindas". (Nov. 8, 1996).
[6] Kerth, Norman, "Leadership from the Technical Side", Pacific Northwest Software Quality Conference Workshop, (Oct. 19, 1996), p. 43.

Having a vision is the first step; sharing the vision is the next. Since this image is exclusive to oneself, the image needs to be conveyed to the group. This can be called the bridge between the mind's eye and reality. The bridge can encompass many things. If I look at the most successful change agents, they are able to make a connection with the targets of the change. Each successful change agent had a unique vision and built a bridge for others to see as well. This bridge allowed people to enter their mind's eye and to make a connection.

Once this bridge is constructed, the change agent needs to articulate and share the vision continuously so that others in the group know the direction the organization is going. The vision must also be continually refined over time.

Persistence

Occasionally everyone encounters bad times. However, successful people know how to overcome the odds and achieve success. Did they get lucky or did they just not give up on themselves? When change agents encounter bad times, they usually develop a variety of creative solutions in order to work around or overcome the roadblocks that they face.

Not every venture is successful. Often ventures seem to fail for no reason. In addition, some plans stumble at various points, but are still considered successful. This translates well into process improvement. I always remind myself of playing football. In any game, it is rare that the winning team did not scored points on each opportunity. To be honest, many successful teams make critical mistakes during the game or start from poor field position. In my opinion, making change follows the same paradigm.

Confidence

Confidence comes from within a person. People exhibit confidence through their actions, composure, and speech. How do people develop this confidence? They know internally that they will achieve their goals. They may not know how, but they believe that they will. This inner energy is what drives them.

Optimism

Optimism can inspire a group to achieve a challenging goal. An optimist lends his or her energy to people that need hope. By keeping hope alive, it will give people the extra push to get through their own personal barriers. Optimism can take many forms. Sometimes cheerful praise or quiet sentiment is what a person needs to help them along their path.

*Change = **?** +Time+Reinforcement + Reaction + State + Culture + Change Element*

Are there more factors involved in the change equation? Probably so, but I struggle to practice well the tools I know before I move on.

Summary

Facilitating change under the best of circumstances is difficult. Facilitating change in the fast changing software world often complicates the situation. From trial and error, I have learned that to be successful in facilitating change you must first understand change. More specifically, that change is an equation of several factors to be introduced at different times. Sometimes the equation is a simple one; sometimes the equation is very complicated. There is just no simple answer. I often think of what Gerald Weinberg wrote, "There is no silver bullet, but sometimes there is a Lone Ranger." [7] People seem to seek out a silver bullet to solve their problems. Unfortunately, there are no silver bullets. Only some normal ones. When these bullets are used at the correct time and in the proper order, real software improvement is made. The trick is to know when the time is right and in what order to proceed. That is why you need a Lone Ranger, or a successful change agent.

[7] Weinberg, Gerald M., Quality Software Management, Congruent Action, 3, (New York:Dorset House,1994), p.1.

Bibliography

Belasco, James A, Ph.D. *Teaching the Elephant to Dance : The Manager's Guide to Empowerment.* New York:Plume, 1991, p.23.

Bindas, James. R. "Meeting Minutes between Neal Brenner and Jim Bindas". (Nov. 8, 1996).

Bindas, James. R. "Meeting Minutes between Dale Emery and Jim Bindas". (Nov. 9, 1996).

Kerth, Norman. "Leadership from the Technical Side." Pacific Northwest Software Quality Conference Workshop. (Oct. 19, 1996). p. 43.

Kubler-Ross, Elisabeth, "On Death & Dying". Hudson River Editions Ser:Macmillan, 1993

Weinberg, Gerald M. Quality Software Management, Congruent Action. 3. New York:Dorset House,1994. p.1.

Using Web Browser Technology for Documentation Storage and Retrieval

Thomas E. Canter
Attachmate Corporation
Enterprise Solutions Group
3617 131st Ave. SE
Bellevue, WA 98006

I. Introduction

What we now call the Internet was originally developed as a research tool. Since the Internet became accessible to individuals and companies, its commercial value for advertising, sales, and entertainment has been exploited. Access to many commercial resources has expanded rapidly because of graphical Web Browsers, such as Netscape Navigator®, Microsoft® Internet Explorer, or NCSA Mosaic.

However, the potential of web-based applications for documentation storage and retrieval is a relatively new use of this technology. In the past, many computer users were barred from sharing information because of differing operating systems, system complexity, and hardware costs. Web browser technology can facilitate the development of systems and applications. The Web is a potent research and development tool, and provides a unique opportunity for collaboration in developing operating systems, networking products, or applications.

Creating a virtual storage method for documents in an online database allows dynamic and elegant solutions to file access and relevancy. Document searches can use current Web-based architecture. Web browsers, such as Netscape Navigator® or Microsoft® Internet Explorer, allow simple and effective representation of the document system as related links. Browsing the document directory structure returns a virtual directory structure relevant to the viewpoint of the user.

Using the test documentation collected by Attachmate Corporation as an example, this paper discusses some of the issues in implementing web browser technology to provide access to and organization for that documentation.

1997 Attachmate Corporation. All Rights Reserved

II. Background

A. Attachmate Corp. has amassed an extensive collection of test documentation.

This test documentation spans several years of developing and testing products for various operating systems, including DOS, Windows, Macintosh, UNIX, and IBM mainframes. This collection includes more than 31,000 documents, ranging from historical documentation to current work-in-progress. The relevance of stored documentation to current documentation needs is unknown. Even though product requirements and product features change, much of the stored documentation may be useful.

Current documentation reuse is extremely low because of lack of knowledge of storage location, nonstandard formatting, and unwieldy storage methods. Storage technology has changed with the growth of technology and hardware and software acquisition. Server hardware and software are more sophisticated. Because developers, product managers, and other users can't find out what already exists, they have to start "from scratch" every time they start documenting.

B. Current documentation storage system is fractured and complex.

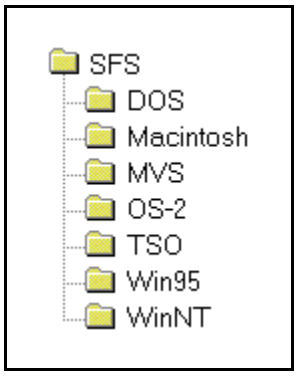
1. File storage organization and structure is haphazard. Documentation is not centralized. It is stored over numerous file servers, sites, and types of servers.

    a. A project uses a dedicated server for the development group. A directory structure that generally mirrors the functionality of the current project is built on the same dedicated server. Development software, special device drivers, OEM software, and administrative information are also stored on this dedicated server. File duplication may reach 25 percent of all storage.

    b. Documents are stored in a project directory, usually under the internal name of the project. The Macintosh support group places their documents in the Macintosh directory, and the Windows NT group places their documents in the WinNT directory. Even if these directories are on the same physical file server, different project groups may not be able to locate relevant documents in other directories because of unfamiliarity with storage architecture standards in other support groups. Document storage locations might make perfect sense at creation time, but after project completion may no longer be relevant.

1997 Attachmate Corporation. All Rights Reserved

For example, Attachmate Corp. produces several products that are designed to connect to IBM mainframes from various operating systems or operating environments. In the root directory, a product directory structure might look like the following diagram. The rigid directory structure makes perfect sense to the support technicians because this is how their organization is structured.



Product Directory

Suppose a support technician is looking for information about file transfer methods. A technician working with a customer using Windows 95 will look under the Win95 directory. Unknown to the technician, a similar problem with the Windows 3.1 product was inherited by this Windows 95 release from a flaw in the Windows 3.1 code base. This problem was detailed in the Windows 3.1 product. A work-around is available for the technician, but he does not locate the file because the directory structure does not enable him to easily find related documents in other directories.

A test document describing a file transfer method using a Windows 3.1 product contains information related to the file transfer protocol that is useful for a Macintosh product. This document is stored on a NetWare® file server and is not easily accessible to a Macintosh test engineer. This may result in the Macintosh test engineer writing his own document about the file transfer protocol.

Later, a product manager is looking up information about customer needs in the Windows NT product. In the Macintosh directory, a support technician wrote up a detailed feature request from a customer. Many customers would like this feature implemented, and it would generate a great amount of interest (sales) if implemented. The product manager does not find this document because she is looking in the Windows NT and Windows 95 directories.

c. Physical file systems give a user direct access to specific files. Locating the files contained in a directory is relatively easy. This type of physical file system method is closely tied to each operating system. The directory structure gives a limited means to locate files and provides clues to what information the file contains.

1997 Attachmate Corporation. All Rights Reserved

2. Related documents often have no physical relationship. The directory structure does not allow for differences in reference. Relevant and useful information may not be immediately available. Differing viewpoints directly affect an understanding of structural relationship.
3. Historical arrangement of the documentation, which was logical to the developer, may not be intrinsically relevant to current users. Users in two years may not know the internal or code name of a product. Yet at the time of storage, the code name was the critical name of the directory that stored all the product documentation. This information is lost and generally not recoverable. The option of moving a relevant file from its current location prevents locating the file by historical reference.
4. The directory structure determines only the intrinsic relationship of the documentation. A Test Manager is concerned with the functional structure of a product that may change between releases. The Product Manager is concerned with requirements. A Functional Test Engineer is concerned with the ability to execute the product. Critical common information is lost, misinterpreted, or duplicated due to incompatible storage, editing tools, or lack of knowledge about previously defined areas. Physical relationships are difficult to manage.
5. The documentation is in various formats. For example, an NFS server might not support MS-DOS–compatible 8.3 file names, and its method of mangling the file name may not follow a standard format.
6. No document status is maintained. Whether an individual document is a draft, a copy, or a final report is not apparent.

This project-oriented file system is an example of a single-model, directed graph representation of a data structure. The data structure has a hierarchical structure. Each node can contain additional nodes (directories) or leaf nodes (files). This method is logical and consequently can be easily navigated by the user. This file model allows file system engines to be integrated with the operating system. Tools for file caching, directory caching, and direct file access for file system defragmentation are relatively easy to implement.

1997 Attachmate Corporation. All Rights Reserved

III. Common Documentation Storage Solutions

A. Archiving

Archiving is a direct method of document storage that establishes rigid storage requirements for system users. Existing documents must be evaluated and moved into a specified directory structure. The archivers might need thousands of hours to evaluate the current documentation system and move documents within the directory structure.

Compliance with rigid storage requirements ensures that the system users can locate documentation. Unfortunately, all users of the system may not understand or agree with the arrangement of the documents in the structure. Even with the strictest controls, documents might not end up in the intended location, or might not fall into the defined categories. On the plus side, the documents are available when needed. Once a user locates a document, she is able to locate the document again.

Maintenance of an archive is expensive. A dedicated server or group of servers probably would have to be dedicated to the storage system. Reviewing the document structure and correcting misplaced documentation or the structure would be an expensive task. Because a company typically establishes new documentation in conjunction with new product development, the archive system might have to be reorganized to reflect the new relationships.

Attachmate Corp. would incur extensive cost to reorganize the current storage system. With an archive of over 31,000 documents, it would cost nearly $1,000,000 to reorganize the current document archive into a consistent structure.

> Estimate: 31,000 Documents x 30 minutes/Document x 1 hour / 60 minutes x 60.00 $/hour = $930,000

B. Automatic indexing

Automatic indexing is another partial solution. Initially, a company would install and configure an automatic indexing server. The initial cost of the indexing is low. Indexing occurs automatically. When new documents are added or moved, the system locates them and updates the document index. The documents would be accessible when the users need them.

Unfortunately, when the database is queried, the user might receive many returns that are not relevant to the requested subject. Each search returns many non-relevant hits for every useful document found. Security access to the documents is based upon the Index Server’s security access.

1997 Attachmate Corporation. All Rights Reserved

The documents returned are not directly editable. To access the document for modification, the user must access the document via a file system or other access like FTP. For example, the document web server currently serving Attachmate's documentation archive returns an HTTP link similar to http://SystemTestGroup/DocumentServer/System Test Docs/Localization/Common/Case Dialog Test.doc. This document is actually stored on a Novell server. The network path to this document is \\STG\STG\System Test Docs\Localization\Common\Case Dialog Test.doc. The user must know how to convert from the HTTP address to the network address to modify the document.

How do we get beyond these problems? Indexing engines provide a starting point. Indexing documents leads to text-to-document databases. Key word queries run on these databases allow the user to retrieve semi-relevant lists of documents containing text that matches their queries. However, as noted earlier, this raw search method based on key words returns many non-relevant documents that the researcher is not interested in.

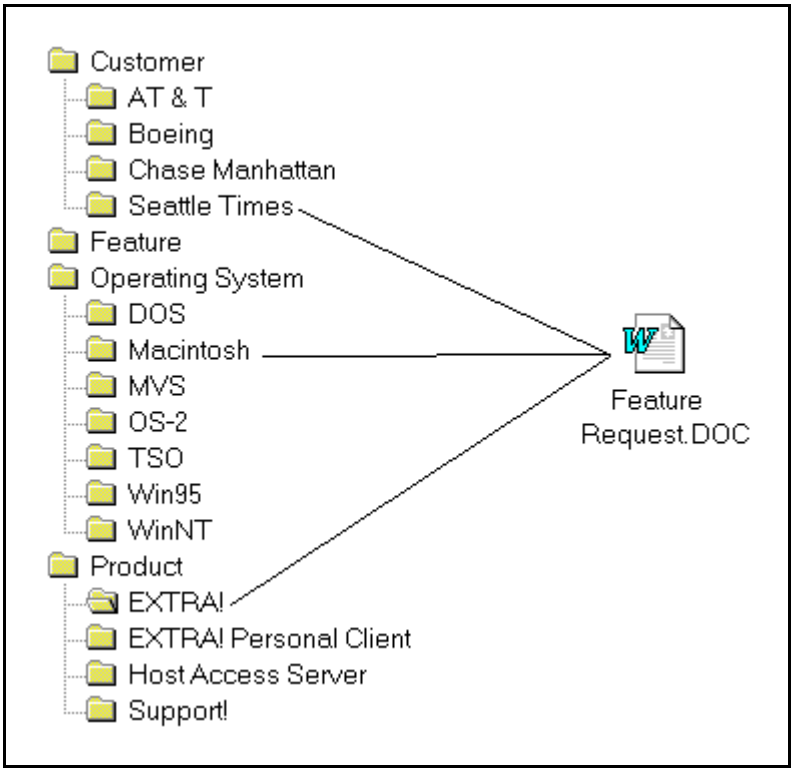
1997 Attachmate Corporation. All Rights Reserved

IV. Creating a Relational Web Server as a Solution

A. A relational document system allows automated management of document relationships.

This relational document system would automatically arrange the hierarchical view of documents based on contents and assigned properties. By isolating the document system from the physical storage that it is designed to emulate, access to related documents is simplified.

Construction of a relational document system allows flexibility. The relationship between file content is mapped to links, so files that contain similar information will appear in the same web page or in linked subpages. Links are virtual and do not exist on any physical file system.



Relational Document System

Documentation storage can be related to the product definition. Product-related documentation can include test documents, requirements, specifications, descriptive documents, test cases, and plans. The storage is extensible to any aspect of the product life cycle including, but not limited to, product support and marketing. <META> tags, such as product revision, author name, product type, date, or any descriptive data, can describe the document relationships. Some of these relationships are described by software test standards. Functional relationships can be used along with execution relationships. Optimizations for performance can be analyzed and options managed at run time.

1997 Attachmate Corporation. All Rights Reserved

Documents can be subdivided into components that are reusable at a smaller granularity. Changes to subcomponents can be selectively applied to all related documents or branched and applied to newly created documents. Projects can inherit documents from previous similar projects. The inherited documents can be branched and modified to exactly fit the current requirements.

Test cases handled online can contain real-time information, such as status (in progress, completed), related defects, links to requirements. Links can be created from test cases to coding elements. For example, a test case that is connected functionally to file transfer can be viewed by a test performer as attached to the file transfer directory structure.

This can be thought of a multi-model, directed graph relationship in which multiple hierarchies are mapped onto the same flat file system.

B. A virtual file system can track the many-to-many relationships between the documents in a directed graph relationship.

Browsing a virtual directory stucture builds virtual paths. These paths are the query key words that the database uses to select documents to display. As each directory is reviewed, the database returns a list of documents that meet the query parameters. Relationships between documents are automatically generated, and can also be managed directly for user relevant viewing.

By selecting query parameters at logon, the user creates a virtual web structure at browse time. A tree-like browser can be used to browse the virtual file system. The file system generated would use standard queries to generate the relative on the fly directory structure. Standard views can be generated and browsed using Java™ or ActiveX™ controls.

1. The first step in creating a relational document structure is to perform an index search on physical directory structures within the organization. This search is similar to the type of index search currently employed by such products as AltaVista or WebCrawler. Documents within the systems are searched to produce a database of returned words. These words are linked to a database of file entries.

   The information contained in each document falls into three general categories:

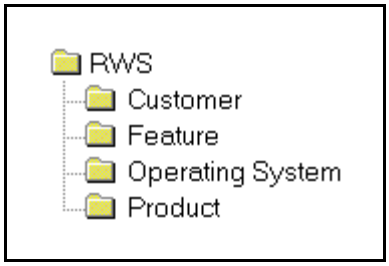
   - Implicit Inclusion. A term of interest is in the document.
   - Explicit Inclusion. A term of interest is not in the document, but a user determines that the document is related.
   - Explicit Exclusion. A term of interest is in the document, but a user determines that the document is not related.

   For example, an administrator at Attachmate Corp. might be interested in creating several first-level hierarchies according to product, features, and customer. Each returned word in the database has a <META> tag identifier. Insignificant words such as and, of, and the are excluded from the database. As the administrator selects entries from the returned word list, each file

1997 Attachmate Corporation. All Rights Reserved

entry that contains a <META> tag identifier created by the index engine is marked with at least one implicit inclusion <META> tag. An implicit inclusion <META> tag means that the document actually contains this <META> tag in its content.

2. The administrator's selections create the following first-level hierarchy. The first-level hierarchy contains words that are arbitrary groupings. The first-level hierarchy in our sample contains the words Customer, Feature, Operating System, and Product.



First-level Hierarchy

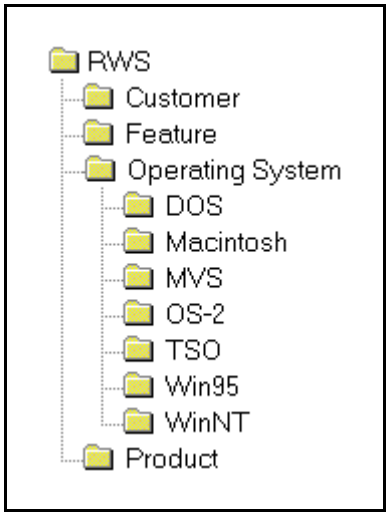
3. By reviewing the frequency of word occurrences in the database, the administrator discovers several product categories that may be of interest to users. A user connecting to the web system discovers this web page created by the administrator.

Product Directory Structure

As the user browses the links, the relational database is performing queries. For example, when the user enters the EXTRA! web page, he sees all documents containing the selected keyword EXTRA! On that web page, he finds the document detailing the feature request by the Macintosh customer.

1997 Attachmate Corporation. All Rights Reserved

4. The same user can browse the following Operating System web page, enter the Macintosh web page, and find the same feature request document on that web page. When the user opens this document, the relational web system arbitrates and opens the file on the remote file system, transferring the file to the user. The user may then modify the document. When the document is saved, the relational web system resolves the document's physical location and stores it on the remote file system.



Operating System Directory Structure

When browsing documents on a relational web system, the user might discover many documents that do not actually belong in a specific directory, even though the index server found the <META> tag within the document. The user would select the document and delete it. Instead of physically removing the document file from the remote file system, the index server only adds an explicit exclusion <META> tag to the file entry. This tag is created in the explicit exclusion table within the entry. When queried, the database discovers this exclusion <META> tag and does not list the file when a search is performed. Instead of deleting the <META> tag from the implicit inclusion table, this method prevents the index engine from continually refreshing the implicit inclusion table and adding the <META> tag back into the implicit inclusion table on the next index pass. By creating a table of explicit exclusion <META> tags, this conflict is prevented from occurring.

The user might enter a directory in which several thousand entries meet the search criteria. He opens several documents and notes that they contain a subset of information for a particular customer or functional area. The user then creates a new web page with that term, for example, "File Transfer". The web system notes the new web page created and queries the returned word database. The <META> tag is added to the implicit inclusion table. Every file meeting the search criteria containing this new <META> tag entry is added. Note that the implicit inclusion table only contains <META> tags of

1997 Attachmate Corporation. All Rights Reserved

interest to the system. As the user enters the new web page, it is already populated with all the files in the preceding web page that contained that <META> tag. If he returns to the parent web page, the selected files will no longer appear. When creating a page, the search engine checks the page definition, and finds all pages that are contained by the page and does not list files that contain the <META> tag defined by all contained pages. Therefore, the page only lists documents that meet the search criteria and yet do not belong to pages that are contained by the page.

The user may discover a document in the parent web page that belongs in the dependent web page, although it doesn't contain the <META> tag that he just created. He moves the file into the web page. The web system notes this and adds the <META> tag to the explicit inclusion table of the file entry. All other web pages defined in the relational web system that contain this file, and that also contain a web page of that same name, will now also display the file in the dependent web page. If the file is open on another web page, then the file would be locked, and the user on the current web page would be prevented from moving the file.

5. Each user has a virtual personal web page and can add new documents.

   As part of the administration of the relational web system, each user receives a virtual personal web page, with an address similar to http://relationalweb/users/username. For example, user Susan Davis creates a new file on her personal web page http://relationalweb/user/Susan Davis.html. After she creates the document, the web system notes the close of the document and unlocks it. When the document is unlocked, the file entry is placed in the index engine's work queue. The index engine opens the document, and adds the document to the appropriate word list entries. The document receives the implicit inclusion tags according to the rules set down by the administrator and users (as they create new directories). The document then becomes available to all users. New documents created or added to the system will be processed by the index engines work queue first.

   It is interesting to note that the user name "Susan Davis" has become an explicit <META> tag because a document was created on her virtual web page. The document inherits all of the <META> tags from the current web page and all of its predecessor web pages. It will appear in any web page with the name "Susan Davis" if it meets all of the predecessor page search requirements.

   The relational web system can create a personal profile of each user, including information about preferences, level of access, remote file system passwords, and other relevant user information. This profile will contain any information necessary to correctly access and control user connection to the relational web system and remote file systems.

   At some point, the user becomes interested in a file that she has found in the Functional web page. To make it easy to locate, she copies the file from the Functional web page to her personal virtual web page. The user's <META>

1997 Attachmate Corporation. All Rights Reserved

tag entry is made in the explicit inclusion table of the file, and the document now appears on the user's personal web page. When the file is deleted, the <META> tag is removed from the explicit inclusion table, and the file disappears from the virtual user web page. It is anticipated that this method will be the primary means of adding documents to the document database. It may be desirable to actually prevent users from adding documents to the web pages outside of their personal web page.

C. Deleting files

Only the system administrator or authorized user can actually delete files from the physical store. If files meet none of the inclusion criteria of the system, they will be put in a temporary store, possibly broken down by major word categories. The administrator can examine these orphan files and attempt to place them in explicit categories or move them to offline storage.

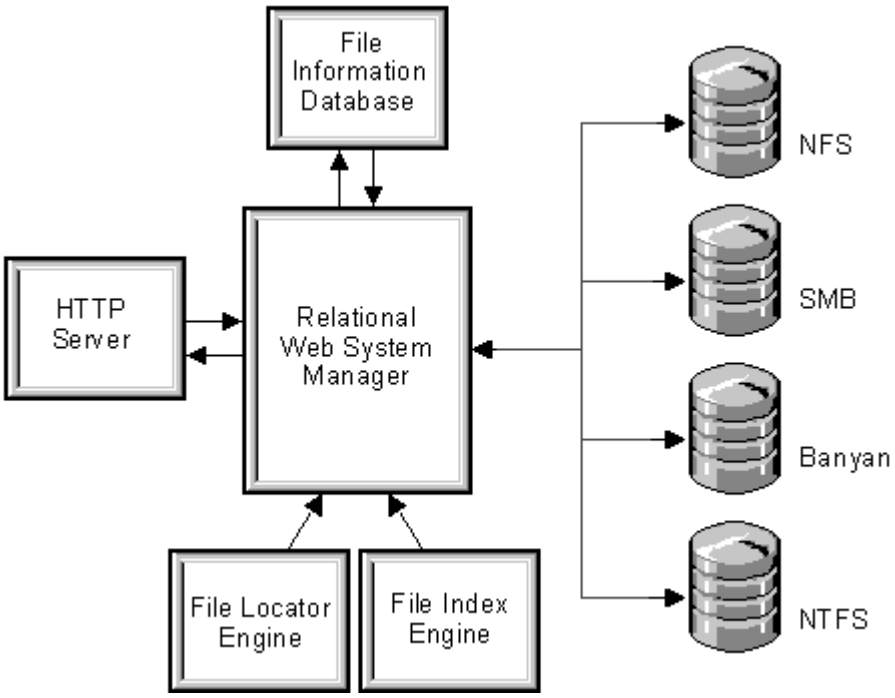
When deleting a web page, the system has two options available. 1) The system may simply remove the virtual web page entry from the current web page. This would make no changes to the explicit inclusion table for files contained by that page. This would allow other web pages displaying the file based upon the same <META> tag to continue to do so. 2) Remove all explicit inclusion tags in all files containing the deleted <META> tag and add in explicit exclusion tags to the files with entries for that word in the Implicit Inclusion table. This recursive method has the impact of excluding the files from other web pages that have that <META> tag throughout the system. The administrator must consider the impact of each option to select appropriate behavior for the environment.

D. File name mangling

It is probable that two file names with the name will end up occupying the same web page. The relational web system generates a unique file name for each instance of the file when it finds a file name clash when the file query for a web page returns duplicate file names. The relational manager would resolve access to the file and the appropriate file would be returned to the user.

1997 Attachmate Corporation. All Rights Reserved

E. A Relational Web Management server consists of several components: an HTTP server, a relational Web system manager, a file information database, a file index engine, a file locator engine, and administrative tools.



Relational Web Server

1. The HTTP Server component performs the following functions:
   a. Managing connections to the client workstations.
   b. Converting the file entry records retrieved from the File System router to client compatible file entries.
   c. Directing all file system requests to the Relational Web System Manager. It extracts the appropriate mangled file name from the file entry record and provides it to the client. If necessary it generates and updates the Arbitrator when a mangled file name must be created.
   d. Gathering security information by prompting clients for storage passwords and limiting access to unauthorized storage systems or locations.
   e. Notifying the Relational Web System Manager when client connections are closed.
2. The Relational Web System Manager performs the following functions:
   a. Creating the client connection table.
   b. Directing directory searches to the File Information Database.
   c. Creating index requests for the index server.

1997 Attachmate Corporation. All Rights Reserved

d. Monitoring file storage requests and directing files to the appropriate remote file system.

f. Identifying pending work requests and prioritizing them for the background processes.

g. Directing file information returned from the file locator and the file server to the correct table in the file information database. It receives notification of found files and prioritizes their indexing based upon administrative rules. It understands file record entries and translates them to remote file system entries. It is a general file router.

3. The file information database performs the following functions:

   a. The word table receives a document, and a list of words and word occurrences from the index server. Nonspecific words such as a, the, and, are edited out by the index server. Each word in the word list is in the column of words in the table. An entry corresponding to the document and the number of occurrences of that word will be added. The document count for that word in the word column of the word table is then incremented. If the word does not exist in the word column of the word table, then word will be added and the document and occurrence data incremented.

      When a document is modified, it is reparsed by the index server and a new list of words and word occurrences is received. If the document is already contained in the parsed documents table, all existing references to that document are deleted. A new entry corresponding to the document and number of word occurrences is entered into the table.

   b. The document table receives the documents when the administrator or a user defines the virtual directory When the administrator selects a word as a directory, that word is added to list of implicit inclusion tags for each document containing that word. If a document is removed from a directory, a tag is added to the list of explicit exclusion tags. The tag is removed from the implicit inclusion list. A document can only be removed from a directory by a direct delete action. If a document is placed into a directory for which it can have no implicit include tag (i.e. the directory “word” is not contained within the document), an explicit include tag can be added for documents that do not contain a defined word is added to the list of explicit inclusion tags.

      Each document will also have a corresponding mangled file name table which will include the file names generated for uniqueness purposes or remote system requirements. The documents will be listed by complete path in a binary field.

   c. The parsed documents table is a list of all documents that have been parsed by the index server and that are contained within the database.

1997 Attachmate Corporation. All Rights Reserved

4. The file index engine retrieves entries from the pending priority work queue and processes them. It extracts words from the found files and updates the master word index.
5. File locator engine is a background process that iterates the remote file systems in a priority queue manner updating the master file list. It can prioritize the file systems based upon the criteria set by the file system arbitrator.
6. Other administration tools are required to maintain the file structure relationships and examine unwanted exclusions, manage security rights, map file system user names to remote file systems, perform the initial database construction, and create remote file system storage access rules. These tools are the keys to maintaining a clean environment for the user. Weak or poorly designed tools will prevent the system from being useful to the administrator and users.

1997 Attachmate Corporation. All Rights Reserved

Standardized Data Representation for Software Testing

David A. Mundie

Introduction

The lack of standardized data representations is a serious problem in many industries, and software testing is no exception. Fortunately, there is a straightforward, proven solution provided by SGML technology.

In this paper I take a quick look at the chaotic situation which I confronted when I first became involved with software testing, and which I think is representative of many parts of the industry. Next I describe the Standardized Testing Markup Language (STML) which my colleagues and I designed in an effort to control the situation we faced. Finally I describe our experiences in using STML to build a Web-based software testing environment.

Anecdote: The Tower of Babel

At a company I worked for recently, there were 35 test suites in use. This is as it should be - the products being tested were large and many-faceted, and many different aspects of those products needed to be evaluated.

What was not as it should be was that those 35 test suites were supported by 33 different test harnesses. For all intents and purposes, every test suite had its own unique set of tools to support test specification, selection, execution, analysis, and reporting.

To illustrate, let us take a look at some of the different test logfile formats that were actually in use in that company. Some of them were nothing more than traces of the execution of the test:

```
;Enter t_func.cmd
;func twice
;func funct1
;func funct11
;func funct22
;func funct33
;func main
;func stop
;dlog master.log,a
```

Some of the log formats attempted to record the environment in which the tests were executed, but without any standard nomenclature. Compare

```
USER         ==  shantz
date         ==  Tue Jun 11 23:34:46 EDT 1996
hostname     ==  tlsund23
tada         ==  /tlsund23_3/develop/install/c40/v5.1/bin/tadac40
adalib       ==  /tlsund23_3/develop/install/c40/v5.1/bin/adalibc40
```

```
TAL_UNIV     ==  /tlsund23_3/develop/install/c40/v5.1/std_packages
library      ==  regression15232
switches     ==  -g
source       ==  /adacore/acvc1_11
acvctools    ==  /pit/acvctools/v5/bin
dlhost       ==  tlsund23
host         ==  /stm/testbeds/sunos/C44_1/acvc/testbed
```

with

```
************************************************************************
                 Version: COMPILER TESTER - VERSION 2.55
               Host Type: SPARC
               Host Name: tlsund22
            test_122.log : Started @ Mon Mar 31 16:24:48 1997
            Command File: armpp02_a.sun4
************************************************************************
------------------------------------------------------------------------
                          OPTIONS USED
------------------------------------------------------------------------
TI PROCESSOR GROUP = arm   TESTS BUILT WITH THESE OPTIONS = -q -o3 -oi0 -mt

CMD FILE  = /db/tip01/swpe/ctest/arm/build/lnk.cmd

CIO NAME = cio_122.lib   CIO OPTS  = -q -o3 -oi0 -mt
CIO PATH = /db/tip01/swpe/port/arm/cio

RTS NAME  = rts_122.lib  RTS OPTS  = -q -o3 -oi0 -mt
RTS PATH = /db/tip01/swpe/port/arm/rts

WORKING DIRECTORY = /db/tip01/swpe/ctest/arm/build/tmp/122
```

and

```
begin
architecture=     TMS470R1X
model=            Texas Intruments
timestamp=  1997 173 1553
compiler=   TMS320C6X CGT v1.10
options=    /db/sds/prodeng/port/c60/product/sun4/cl6x -ft/tmp -O3 -
I/db/sds/prodeng/ctest/nullstone %D %{%T } -z -o stone.100 main.obj
testname=   src/alias/alias_01.c
title=            Alias optimization of short object versus other types.
language=   ANSI C
optimization=     Alias Optimization (by type)
testtype=   Performance
command=    /db/sds/prodeng/port/c60/product/sun4/cl6x -ft/tmp  -O3 -
I/db/sds/prodeng/ctest/nullstone -DS=EXTERN -DT=INT -DO=PLUS -DV=None
src/alias/alias_01.c  -z -o stone.100 main.obj libnull.obj -
l/db/sds/prodeng/port/c60/product/sun4/rts6201.lib -llnk.cmd
compiletime=      5
compile=    pass
rate=       716845
ideal=            no
execution=  pass
```

```
end
```

Perhaps the least standardized feature was the way errors were reported by “self-checking” tests: each test harness designer seemed to delight in inventing new syntactic conventions. Here is just a small sampling, showing 13 different ways of reporting “PASS”:

```
1. #PASSED:  c61a

2. ***** 821 successful test cases in PREC1 *****

3. ++++++++++++P50001.c Passed

4. execution =    pass

5. ==== C34001A PASSED ============================.

6. PASSED 1996-04-30 17:53:58 mundie

7. ********************************
   PASSED ALL REAL ATTRIBUTE TESTS
   ********************************

8. FRACT(LDEXP(x,i)) = x; for 1/radix<=abs(x)<1.0, vemin<=i<=vemax
   PASSED!

9. 220|1 1 0 11:34:00 19941030|PASS

10. --PASSED NEG tests

11. 103|          PASSED ("CONSTRAINT_ERROR NOT RAISED -- Y := E2");

12. -- -2147483648 /          -1 = -2147483648  (NO EXCEPTION)  ok

13. Section s22     returned 0.
```

This lack of commonality had several harmful consequences. Most of the 33 harnesses were of poor quality, since their development costs had to be amortized over a single suite - they were just thrown together to meet the needs of the moment. The learning burden for novice testers was very large, since learning how to use one test suite didn't help in learning how to use the next one. Worst of all, it was very expensive to expand the analysis tools available in the testing environment, because each tool had to “understand” each distinct flavor of logfile. If a new reporting tool was needed, it had to be written 33 times instead of just once.

This situation seems endemic to the software testing industry. It is hard to name two commercial, or even public domain, test suites that use the same test harness. There have been attempts to bring the obvious benefits of standards and interoperability to the field, but they have followed the usual approach of attempting to standardize by gaining market dominance: each purveyor of testing tools has attempted to impose its own proprietary harness as the *de facto* industry standard, but none has succeeded nor is likely to succeed.

Standardizing Document Formats with SGML

To be sure, the software testing industry is not the first to realize that having a standardized document format would permit great savings. One of the classic early examples was the plethora of price list formats in use by the thousands of suppliers for the DOD, but there are dozens of other examples in technical publishing, law, medicine,

semiconductors, and many other fields. It is coming to be understood that what is needed for interoperability is not a standardized toolset, but rather a standardized format for data representation.

The *de facto* industry standard for defining such standardized data representations is SGML, which was used to such astonishing success in defining HTML. SGML has several advantages when it comes to defining document types for software testing. (1) It is an open, international standard, with a substantial user community and accumulated experience. (2) There are many pre-existing SGML tools on the market, including editors, browsers, transformation tools, and database managers. (3) Even in the absence of third-party tools, SGML-based languages have a clean syntax that is easily manipulated by tools such as Gema or PERL. (4) It is designed to allow users to describe the *semantics* of their documents, rather than just their formatting. It is exactly this semantics-oriented approach that is needed for interoperability of testing tools. (5) A streamlined subset of SGML, the Extensible Markup Language (XML), is gaining momentum as the successor to HTML for the next generation of Web markup language. With both Microsoft and Netscape promising to provide browser support for XML, we can look forward to the day when SGML-based documents will be browsable from widely available tools.

Given all these advantages, it seemed natural when my colleagues and I sat down to unify our test environment to use SGML document types as the basis for interoperability.

STML Log Files

SGML document types are defined in Document Type Declarations (DTD), which define the various kinds of text, called *elements* in SGML, which make up the document. Each element has a corresponding "tag" which is used to mark the element within the document. For example, the name of the test being executed in the STML log file is indicated using the opening and closing tags "<NAME>" and "</NAME>".

Currently STML defines three main document types: logfiles, test databases, and test result summaries. The most important of these document types is the test logfile, because it plays such a central role in the testing process. It is first and foremost the standardized logfile format that allows interoperability of testing tools.

The STML logfile is intended to capture all the information from one test run that could possibly be of interest to subsequent analysis. It is best thought of as a small textual database recording the environment and result of the test run, although because it uses SGML it is a flexible, extensible database that can handle partial data and does not impose a rigid fixed-field database schema.

Here is a sample STML logfile:

```
<LOG>
  <HEADER>
    <DATE>1996-09-23 11:17:34</DATE>
    <VERSION>1.1</VERSION>
    <USER>mundie</USER>
    <COUNT>2</COUNT>
    <VAR>host    tlsund2</VAR>
    <VAR>target  c30</VAR>
  </HEADER>

  <PHASE kind="build">
    <NAME>test1</NAME>
    <START>1996-09-23 11:17:46</START>
    <CAPTURE>
      Acme compiler v 0.0
      Compiling test1.d.
      Compilation successful.
    </CAPTURE>
    <FINISH>1996-09-23 11:17:53</FINISH>
    <RESULT>PASS</RESULT>
  </PHASE>

  <PHASE kind="execute">
    <NAME>test1</NAME>
```

```
    <START>1996-09-23 11:17:46</START>
    <CAPTURE>
      Test 1 - check that 2+2=4
      *** Success!! 2 + 2 = 4
    </CAPTURE>
    <FINISH>1996-09-23 11:17:53</FINISH>
    <RESULT>PASS</RESULT>
  </PHASE>

  <TRAILER>1996-09-23 11:18:15</TRAILER>
</LOG>
```

As can be seen from this log, a logfile consists of a header, one or more test phases, and a trailer. The header records the date and version of the tests, the user, the expected number of test phases, and a description of the test environment by means of the "Var" tags.

The test phases are delimited by "Phase" tags. STML does not predefine any phase types, but instead requires the user to specify a "kind" attribute to mark the phase as a build phase, an execute phase, or any other type of phase that might be appropriate.

Each phase marks the time the phase begins and ends, and captures the output of the phase. It also records the result of the test.

This structure is nicely expressed in the document type definition for STML logfiles:

```
<!DOCTYPE LOG [

<!ELEMENT LOG      - - ( HEADER, PHASE+, TRAILER )>
<!ELEMENT HEADER   - - ( DATE, VERSION, USER, COUNT, VAR* )>
<!ELEMENT PHASE    - - ( NAME, START?, CAPTURE, FINISH?, RESULT )>
<!ATTLIST PHASE
                     kind  CDATA #REQUIRED>

<!ELEMENT TRAILER  - - (DATE)>
<!ELEMENT DATE     - - (#PCDATA)>
<!ELEMENT VERSION  - - (#PCDATA)>
<!ELEMENT USER     - - (#PCDATA)>
<!ELEMENT COUNT    - - (#PCDATA)>
<!ELEMENT VAR      - - (#PCDATA)>
<!ELEMENT NAME     - - (#PCDATA)>
<!ELEMENT START    - - (#PCDATA)>
<!ELEMENT CAPTURE  - - (#PCDATA)>
<!ELEMENT FINISH   - - (#PCDATA)>
<!ELEMENT RESULT   - - (#PCDATA)>
]>
```

The content for each element is given using a regular-expression-like syntax. Thus

```
<!ELEMENT LOG      - - ( HEADER, PHASE+, TRAILER )>
```

means that a log consists of a header, one or more phases, and a trailer. The “- - “, by the way, means that neither the opening nor the closing tags may be eliminated. This is done for compatibility with XML, which does not allow markup minimalization. The “#PCDATA” symbols are SGMLese for “any old text”.

Finally, the “ATTLIST” declaration specifies that PHASE elements must have a “kind” attribute whose value is just character data.

The advantage of having the document types defined with a formal grammar like this is that document instances can be validated, guaranteeing that they conform to the declaration and consequently can be reliably processed by other tools that work off the same DTD.

Test Databases

Another common requirement for test automation is a database of information about the tests in a given test suite. Such information includes the expected execution time of the test (for use in setting timeout values), special requirements for building the test, and information about what features the test exercises.

Often this information is scattered in the scripts that are used to execute tests, or is encoded in multiple directory structures; at best it is stored in an actual database. SGML is ideal for storing this type of information in a flexible, transportable and efficient way. In particular, it is possible to establish defaults for the suite as a whole, and only override those defaults where needed, since typically only a few tests will need to have special values recorded.

In the interest of brevity I shall not give the DTD for the STML test database, but here is a short section of an example database:

```
<SUITE>
   <NAME>PIWG

   <CUSTODIANS>mundie ciccarel villani egan</CUSTODIANS>

   <EXECRULES>Execute.rules</EXECRULES>

   <TARGET kind="1750a">
      <LCF>1750aDefaultlcf
   </TARGET>

   <TARGET kind="c30">
      <LCF>c30Defaultlcf
   </TARGET>

   <TARGET kind="c40">
      <SWITCHES> -Op2 -g
   </TARGET>

   <TIMEOUT>900</TIMEOUT>

   <TEST>
      <NAME>a000001</NAME>
      <ATTRIBUTES>smoke all compile foundation nolink</ATTRIBUTES>
   </TEST>

   <TEST>
      <NAME>a000011</NAME>
      <TIMEOUT>1800</TIMEOUT>
      <SWITCHES> -Op3 -A
      <ATTRIBUTES>all compile runtime nolink</ATTRIBUTES>
   </TEST>

</SUITE>
```

This fragment describes the PIWG test suite, whose maintainers are given in the "CUSTODIANS" element. The rules for determining whether a test has passed or not are specified to be in the file "Execute.rules". Two target-

specific linker control files are needed, and on the C40 target a different set of switches is needed. The default timeout for the suite as a whole is 900 seconds.

The entry for each test in the test suite contains the test's name and a set of attributes. These attributes may be used, for example, to extract subsets of the tests for special purposes. Each test entry may additionally override the defaults for the test suite; for example, test a000011 needs 1800 seconds to execute, and requires a special set of switches.

Placing all the test-suite-specific information in a single structured document greatly simplifies its maintenance and makes it available to the tools that need it in an efficient way.

Test Summaries

A third common requirement in testing is a database of test result summaries, useful for tracking project progress and for many other purposes. STML defines the TestSum element for this purpose. Here is an extract from an example project database:

```
<TESTSUM>
  <HEADER>
    <PROJECT>C40

    <TOTALS>
      <SUITE>CAMP
        <PHASE kind="Build">380
        <PHASE kind="Execute">380
      </SUITE>
      <SUITE>SPR
        <PHASE kind="Build">0
        <PHASE kind="Execute">125
      </SUITE>
    </TOTALS>
  </HEADER>

  <REPORT>
    <Test-SUITE>CAMP
    <REPORTED>
      <DATE>1996-03-17 11:03:36
      <BY>mundie
    </REPORTED>
    <Level>Routine
    <RUN kind="Build">380
    <PASSED kind="Build">375
    <RUN kind="Execute">375
    <PASSED kind="Execute">373
    <COMMENTS>
    Had to restart due to power outage.
    </COMMENTS>
  </REPORT>

  <REPORT>
  ...
  </REPORT>

  ...
</TESTSUM>
```

The header contains global information about all the test summaries: the name of the project for which they were run and the number of tests in each phase of each test suite. The remainder of the summary document consists of individual test reports. For each test, the date and the name of the person reporting the results, the test cycle, and the number of tests that were run and the number that passed are recorded for each test suite.

From such a database, it is straightforward to extract graphs of the passed/run percentage over the life of a project. It is also important to note that this database can itself be extracted automatically from the relevant STML logfiles.

Experience Using STML

STML was in use for nearly a year before the company developing it ceased to exist. During that period, the ten most important of the 35 test suites mentioned above were integrated into the STML environment. This integration was accomplished in two ways. Wherever possible, the test suite itself was modified to generate STML logfiles and to use STML test suite databases. When converting the test suite proved impractical, postprocessors were written to convert the original logfile formats to STML.

A number of STML-based testing tools grew up in this environment. A Tcl-Tk-based graphical user interface used the test suite databases to provide point-and-click control of test selection and execution. Several different report generators queried the STML logfiles to analyze failures, timing data, and so on. AWeb-based test result browser used Common Gateway Interface applications to down-convert the STML documents into browsable hypertext documents that allowed the user to jump, for example, from the error summary to the point in the logfile where a given error occurred. Finally, forms-based utilities allowed testers to generate and display STML test result summaries.

Although no quantitative data are available, I think it is safe to say that more test analysis and reporting tools were written during the year of STML use than during the preceeding 10 years of the company’s testing history, and that this was directly attributable to the fact that each tool had to be written only once, not once for each test harness.

Conclusion

The STML DTDs in their current form are rather primitive, providing only the most basic information needed in a software testing environment. However, by adding new elements and attributes, they can be easily and cheaply extended to fit the needs of any particular project, and they show promise as a way of leveraging test tool development through greater interoperability.

AN ANALYSIS OF DIAGNOSTIC INCONSISTENCIES IN ANSI C COMPILERS

Gregory Hall
Department of Computer Science
Southwest Texas State University
San Marcos, TX 78666
ghall@cs.uidaho.edu

Paul Oman and Ben Colborn
Software Engineering Test Lab
University of Idaho
Moscow, ID 83844-1010
oman@cs.uidaho.edu

ABSTRACT

ANSI C stipulates that a conforming compiler must generate a diagnostic message for any violation of the standard. However, the severity of this diagnostic, warning or error, is left to the compiler vendor. This paper presents results of a study into the variability in the classification of some violations. This diagnostic variability can impact the portability of source code from one compiler to another, or even to upgrades of the same compiler.

Keywords

Diagnostics, ANSI C, Portability, Compiler design

BIOGRAPHY

Gregory Hall is an assistant professor of computer science at Southwest Texas State University. He obtained his Ph.D. in computer science at the University of Idaho. His interests include software engineering, software measurement, reliability modeling, and software evolution. He has worked on research projects involving Hewlett-Packard, Lexmark International, Northern Telecom, and Jet Propulsion Laboratories. He is a member of the IEEE, the IEEE Computer Society, and the IEEE Test Technology Technical Committee.

Paul W. Oman is an associate professor of computer science at the University of Idaho, and an independent software consultant who specializes in software analysis. His consulting practice includes contract work for several international corporations and U.S. government agencies involved with the construction and maintenance of software for power production, medical instrumentation, environmental control systems, and personnel tracking and utilization. He occupies the Hewlett-Packard College of Engineering Research Chair at the University of Idaho, where he is director of the Software Engineering Test Lab. He has a Ph.D. in computer science and is a member of the IEEE, IEEE Computer Society, and ACM.

Ben Colborn is a research assistant at the University of Idaho's Software Engineering Test Lab, where he works as a technical editor and network administrator. His projects have included metrics analysis of large software systems and research in computer-mediated distance education. He will receive his B.A. in English in May 1998.

INTRODUCTION

Software portability has been the focus of a great deal of research in the software development community. With the rate at which new hardware is being developed and architectures are changing, the ability to reuse software components in new environments is a must for companies that want to maintain a competitive edge [Jaeschke89, Ryan92]. Much of the research into software portability has focused on the development of guidelines for developing portable software. Other research has investigated the potential for software metrics to ascertain the portability of software modules. These efforts have focused primarily on the theoretical concept of portability, trying to answer the question, "what does it mean for a software module to be portable?"

Our initial research goal was to examine portability from a more empirical standpoint. For a software developer, a system is portable if it can be transferred to a new environment and perform its expected function [Rabinowitz90]. The amount of effort and code change required to accomplish this task can be used to assess portability of a module or system. One indication of the amount of effort that may be required is the set of compiler diagnostics that are generated after a port. Having these diagnostics before porting the software, however, would be a great aid to the engineers. The initial goal of this research was the creation of a compiler simulator. Without moving code or buying new compilers or machinery, the simulator would generate the diagnostics that would be presented if the port was attempted. As new compilers were created, their diagnostics could be added to the existing set. A user would provide three items—a set of source files, target compiler and target environment—and the simulator would show any diagnostics that would result from a port attempt. This is similar in nature to many lint-like tools, but the results would be specific to a particular compiler.

The process was to first obtain a set of compiler diagnostics for a particular compiler, write code that would violate them all, and then write a tool that would produce the appropriate diagnostics from this code set. It was believed that once an initial compiler was implemented, adding other compilers would be a simple matter of producing the appropriate message once a particular code construct was detected. Our initial compiler was the GNU C compiler, gcc. The source code to gcc was scanned, and the error and warning messages were extracted. Test code samples were then written to exercise each of the conditions described by the messages. It soon became apparent that some messages were only accessible when certain compilation options were passed to the compiler. In order to handle this situation, we narrowed our focus to only messages that could be obtained when performing a compilation in conformance to the ANSI C standard [ANSI/ISO90]. Once a large set of code samples was created, we then processed the code with a different compiler, the cc compiler of HP-UX. Out of the 175 test cases of bad code, 51 cases did not provide consistent diagnostics. Errors in gcc were not always errors in cc. The same piece of code could compile and run under one compiler and completely fail to compile with the other.

Table 1 presents an example of some results obtained from submitting the same code to various popular compilers. Of the 175 violations tested, there were 25 violations identified as outliers for reasons discussed later in "Analyzing the Information." Without the outliers, 78 of the violations were classified as being consistent across compilers and 97 were inconsistent. That means that approximately 55% of the code samples tested were treated differently by different compilers. With the outliers, only 60 violations were handled consistently by all compilers tested, leaving 115 inconsistent. So, with the outliers the percentage of samples handled differently jumps to 66%. In either case, more than half of the erroneous code samples are treated differently by different compilers. This variability exists among different compilers in the same environment (operating system and machine architecture), different compilers in different environments (varying operating systems or machine architectures), different

versions of the same compiler, and the same compiler in different environments. This paper presents our findings regarding the variance in compiler diagnostics.

ID	Sample	gcc	cc	bcc	mc	ic960
1004	long long long is too long for GCC	E	E	E	E	E
1006	ANSI C does not support long long	W	W	E	E	N
1007	ANSI C forbids zero-size array	W	E	E	E	N
1013	conflicting types for built-in function	W	N	N	N	N
1038	character constant too long	E	W	E	E	E
1056	no semicolon at end of struct or union	W	P	E	P	P
1057	pointer value used where a floating point value was expected	E	E	E	E	E
1058	ANSI C forbids newline in string constant	W	E	E	E	N

E=error, W=warning, P=parse error, N=no diagnostic

Table 1. Example diagnostic variability across popular compilers

An earlier phase of this research is documented in [Pearse96]. The focus of that paper is behavior undefined in the ANSI C standard; that is, conditions that the standard allows the compiler implementation to define. A set of programs exercising ANSI C undefined behaviors were used to test 5 ANSI-compliant C compilers. Of these programs, 76 failed to port to at least one other compiler due to undefined behaviors. This paper, by contrast, is not limited to undefined behaviors, but deals with all violations of the ANSI C standard. A total of 14 different compilers (taking into account different compilers and different versions of the same compiler on the same platform and on different platforms) were tested with 175 direct violations of the ANSI C standard for this study. The important observation is that compilers diagnose 66% of these violations inconsistently.

OBTAINING INFORMATION

The first step in assessing the different compiler diagnostics capabilities of C compilers was the determination of a base set of diagnostic messages. We use the term “violation” to describe the varying diagnostic messages, because compilers do not consistently apply the terms “error” and “warning.” One compiler might classify a code condition as an error while another would classify it as a warning. For that reason we have chosen the term “violation,” and use it to refer to programming constructs that are either in violation of the language definition or that represent questionable uses of language features. The core set of violations that we began with were the diagnostic messages generated by gcc. The source code for gcc was examined and error and warning messages were extracted. While the set of diagnostic messages we obtained is not necessarily the complete set of messages that can be generated by gcc, roughly 250 violation messages were identified. These messages served as our base set of violations, and each was assigned a unique identifier.

The second step was to produce code samples that would generate the appropriate diagnostic message when submitted to the gcc compiler. It became apparent that many of the messages we had extracted from the source code could only be witnessed when certain compilation options were passed to the compiler. Code that blatantly violated the condition expressed by a message would often receive a different diagnostic message from different compilers. For that reason we had to select a base set of compilation options that could be used to focus our efforts. We were not interested in additional language features or platform dependent features of programming languages, since those features are often inherently not portable. We chose to use the ANSI C compliant mode for the gcc compiler with the pedantic option set. This would excise diagnostic messages that were violations of the ANSI C standard

and would disallow any gcc language extensions. This capability was also available in the other compilers that we were interested in evaluating. Code samples were eventually written for 175 violations of the ANSI C language definition.

Each code sample was submitted to gcc and its resultant diagnosis was recorded in a table. The table consisted of rows for each violation, using the identifiers assigned earlier, and a column for the gcc compiler. The result of the compilation for each code sample was recorded in the table next to the violation number of the violation that it exercised. It was not always possible to exercise only one violation. In some situations, a warning would be given related to a suspicious declaration and an error occurred during the definition. The warning was unavoidable and was related to the fact that a subsequent definition would be erroneous. Whenever possible, however, code samples were designed to expose only one violation.

After submitting all 175 erroneous programs to gcc, the same set of code samples was then submitted to cc. A new column for cc was added to the table and the compilation results were recorded. In the case of the newly added compiler, the messages provided were not exactly the same as those for the gcc compiler, from which the violation set originated. Messages had to be compared to determine if they referred to the same violation. The semantics of the different messages were what needed to be compared. For that reason, the process of adding compilers to the matrix could not be automated. Two research assistants manually performed the comparisons by examining the compiler output individually and updating copies of the table. The two copies were then resolved and discrepancies between the human experts were discussed and resolved. This process was repeated for all compilers listed in Table 2.

Compiler	Version	OS	#W	#E	#N
gcc	2.4.5	HP-UX 9.05	74	95	13
gcc	2.6.3	Linux 1.2.8	72	96	14
gcc	2.7.2	HP-UX 10.10	73	96	13
cc	9.05	HP-UX 9.05	38	117	27
cc	10.10	HP-UX 10.10	39	118	25
CC	9.05	HP-UX 9.05	12	125	45
ic960	2.6.0	HP-UX 9.05	49	95	38
Borland C++	4.02	DOS 6.22	21	136	25
Borland C++	2.0	OS/2 3.0	23	134	25
Borland C++	5.01	Windows 95	21	137	24
Microsoft C/C++	7.0	DOS 6.22	22	127	33
Microsoft C/C++	10.0	Windows NT 4.0	23	123	36
Zortech	3.0	DOS 6.22	1	128	53
Watcom	10.0	OS/2 3.0	20	132	30

Table 2. Compiler list

In some instances, the simple distinction of a violation as a warning or error was insufficient. Due to the differences in parsing capabilities, a particular code sample would cause parse errors on some compilers, while other compilers would parse the code and potentially identify a particular violation. Such instances are represented in the table by the letter P, for parse error. Also, some compilers generated warnings about a particular violation, but that code sample would cause a subsequent linker error. The matrix was augmented in these cases with an additional value, in parentheses, which indicated how the violation was ultimately treated. For instance, a warning that was generated for a particular violation but caused a subsequent compilation failure would be given the value W(E). Other notations were also needed. For

example, one code sample generated a message that was tagged as an error, but was ignored. The notation E(i) represents a violation for which the compiler produced an error message but ignored the error. This is semantically equivalent to a warning.

While this augmented notation was more informational, it made the table more difficult to analyze. For analysis purposes, the possible values for the table were collapsed to either Nothing (no error or warning message was produced for this code sample), Warning (a message was produced, but an executable was still generated) or Error (a message was produced and no executable resulted). In this version of the table, the value in parentheses took precedence. Errors that were ignored were classified as warnings. This new version of the table, with only three potential values for each of the cells, could be much more easily examined for variability. Violations that were treated uniformly by all compilers are easily identified. These consistent violations were almost entirely treated as errors by all compilers and do not impact portability. If no compiler can generate an executable from the code, then there is nothing to port.

Aside from consistently handled violations, there were three other categories identified based on the amount of variability a violation exposed. A violation was categorized as having low variability if it caused a warning on some compilers but nothing from the others. In this case, the violating code always resulted in an executable. While it is not necessarily true that the behavior of the generated executables is consistent, the code does compile. Most of these violations relate to programming practices that are generally deemed dangerous, such as assignments between different types without a cast.

The medium variability category includes violations that are treated as a warning by some compilers and as errors by the rest. In this situation, an executable may or may not be generated, but all compilers produced a diagnostic warning the programmer of a potential problem. Again, the behavior of the executables may not be consistent. These violations are considered more severe than the ones in the low variability category because code that previously compiled and ran may fail to compile at all on a different compiler.

The high variability category includes violations that are classified as warnings by some compilers, errors by others, and nothing by the rest. This category represents the greatest porting problem because code that compiles and runs on one compiler may fail to compile at all on another compiler. Keep in mind that the code samples being submitted to the compilers all represent bad code or bad coding practice. These samples may draw on implementation defined aspects of the C programming language which can vary between compilers and still meet the ANSI specification. However, since some compilers produce no diagnostic for these violations, there is no way of knowing that a problem exists until an attempt is made to port the code.

ANALYZING THE INFORMATION

The first step in analyzing the table was to dismiss the consistencies. If all of the compilers treat a violation the same way, then the result of compiling the code on a different compiler can be anticipated. Here, the result in question refers to the diagnostics generated by the compiler, not the behavior of the generated executable. The next step was the categorizing of the remaining violations as either low, medium, or high variability. This process was complicated by the existence of borderline cases which we deemed to be outliers. Outliers, in this context, were violations that would be categorized as being more variable based on the value of only one compiler. In most cases, this compiler was either Zortech or CC.

Zortech 3.0 is a rather outdated version of that compiler, and for some violations the value recorded for Zortech would cause the violation to be categorized at a higher level of variability. In these cases, we

categorized the violation based on the amount of variability among the other compilers. The CC compiler under HP-UX was tested to assess its diagnostic capabilities and the ability of a purely C++ compiler to accept standard C code. In some cases, the C++ compiler did not implement certain language features, and often violations generated no diagnostic whatsoever, even when all the other compilers generated an error. Violations were categorized both with and without the values of the outliers.

High variability violations

The violations that will have the greatest impact on the portability of code are those that exhibit a high degree of variability in the table. Table 3 presents some example observations from the full table for the most recent versions we had for the different compilers. From this small sample it can be seen that the variability cannot be immediately attributed to only one or two compilers, but is spread among them all. This observation is confirmed in examining the entire table. While some compilers did exhibit more variability than others (especially true of the outlier compilers Zortech and CC), no compiler was consistent with another.

Violation	gcc-2.7.2	cc-10.10	CC-9.05	ic960	BCC-5.01	MSC-10.0	Zortech	Watcom
1008	W	W	E	E	E	N	E	W
1011	W	N	N	N	E	N	N	E
1028	W	E	N	N	E	P	N	E
1037	W	N	N	N	E	N	E	E
1041	E	W	E	E	N	E	E	N
1043	W	W	E	N	E	N	N(W)	E
1070	E	E	NI	N	P	E	N	N
2061	W	E	N	N	E	P	N	E
2086	W	P	E	N	E	P	E	E

Table 3. Examples of high variability violations

Without outliers, there were 29 violations identified as being highly variable. That amounts to approximately 30% of the inconsistent violations being classified in this group. With outliers that number jumps to 51 and the percent of violations in this group becomes 44%. With three categories of violations (high, medium and low), this category accounts for roughly a full third of the inconsistent violations. Since inconsistently handled violations are more common than consistently handled ones, and the number of highly variable violations is so large, identifying and removing this class of violations can have a tremendous positive impact on the portability of software systems.

Violation 1008 is an example of a highly variable violation. It has to do with the range of acceptable values for enumerated lists. Enumerated lists, according to the ANSI standard, are restricted to the range of the integer type. The C source code statement `enum {A=99999999999 };` triggers violation 1008 and sometimes causes the following statement to be issued: "ANSI C restricts enumerator values to range of int." Submitting this code statement in a C program to the gcc compiler results in this warning. The cc and Watcom compilers generate such a warning. However, the same code results in a compilation error when submitted to CC, ic960, Borland and Zortech. The Microsoft compiler did not produce any diagnostic related to this statement. Source code segments and diagnostic messages for the high variability violations referenced in Table 3 can be found in the appendix.

Medium variability violations

Violations of medium variability are those that generate a diagnostic on all compilers, but the severity of the diagnostic varies. The code samples in this set result in a warning or error on every compiler, but may or may not result in an executable being generated. These violations are considered to be less dangerous to portability since there is a diagnostic identifying the potential problem generated by each compiler tested. For the compilers that only produce warnings, the resulting executables may or may not perform identically. In many cases they do not. Pearse has performed research into the run-time behavior of a special subset of these violations [Pearse96], those that are related to annex G of the ANSI C standard. This annex deals with implementation defined aspects of the C language. His research has revealed that the ANSI standard allows compiler writers some flexibility in the creation of ANSI C compilers and that these freedoms can result in conforming programs with different behavior.

Violation	gcc-2.7.2	cc-10.10	CC-9.05	ic960	BCC-5.01	MSC-10.0	Zortech	Watcom
1032	W	E	E	W	E	E	E	E
1038	E	W	E	E	E	E	E	E
1045	W	E	E	W	W	W	E	W
1047	W	W	E	E	E	E	E	E
1080	E	W	E	E	W	W	E	E
2021	W	W	E	W	E	E	E	E
2031	E	E	W	W	E	W	E	E
2062	W	E	W	W	E	P	N	E
3042	W	E	E	W	W	W	E	E

Table 4. Examples of medium variability violations

Table 4 presents a sample of observations that are classified as being of medium variability. With outliers removed, the medium variability category accounts for 29 of the inconsistent violations, roughly 51%. With the outliers the number of violations in the category increases to 52, but accounts for only 45%. This is due to the fact that the number of inconsistent violations increases when outliers are considered. The number of violations in this category indicates that they are just as prevalent as the high variability category. Since they reflect instances where porting may or may not result in an executable, and the behavior of that executable may be in question, these results indicate that cleaning up warnings in code can help tremendously with code that may eventually be ported. Source code segments and diagnostic messages for the medium variability violations in Table 4 are provided in the appendix.

Low variability violations

Low variability violations are those that may or may not result in a diagnostic but always provide an executable. The warnings generated may indicate that a compiler assumption is being made. This relates back to the implementation defined aspects of the ANSI standard. Not all compilers warn when an assumption is made or an implementation dependent feature is used. The resulting executables from different compilers may then vary. Table 5 presents some examples of this class of violation. While these violations do not have as dramatic an impact on the portability of code, since an executable is created in each instance, they do contribute to the conclusion that the varying diagnostic capabilities of compilers can have a tremendous impact on portability.

Violation	gcc2.7.2	cc10.10	CC9.05	ic960	BCC5.01	MSC10.0	Zortech	Watcom
1039	W	W	N	W	N	W	E	N
1059	W	W	N	W	N	N	N	W
2010	W	W	N	W	W	N	N	N(W)
3036	W	W	N	W	W	N	N	W
3037	W	W	N	W	W	N	N	W
3038	W	W	N	W	W	N	N	W
3039	W	W	N	W	W	N	N	W
4019	W	N	N	W	W	N	N	N
4037	W	W	N	N	N	N	N	N

Table 5. Examples of low variability violations

The low variability violation category accounts for the fewest number of inconsistent violations. With outliers removed, there are 19 low variability violations, accounting for about 20% of the inconsistent violations. With outliers left in, only 12 low variability violations are identified and this category accounts for only 10% of the inconsistent violations. Not only are inconsistently handled violations more prevalent than consistently handled ones, but the high and medium variability categories account for 80 to 90 percent of the inconsistent violations. Source code segments and diagnostic messages for the medium variability violations referenced in Table 5 are provided in the appendix.

Categorizing variability

The amount of inconsistency attributable to each level of variability (high, medium or low) can be assessed by dividing the number of violations at a specific level of variability by the total number of inconsistently handled violations. Figure 1 shows the percentage of inconsistently handled violations accounted for by each of the three categories of variability. These percentages were calculated including the outliers described previously. From this figure it can easily be seen that the low variability violations, those that have the least impact on portability, account for a very small percentage (approximately 10.4%) of the inconsistently handled violations.

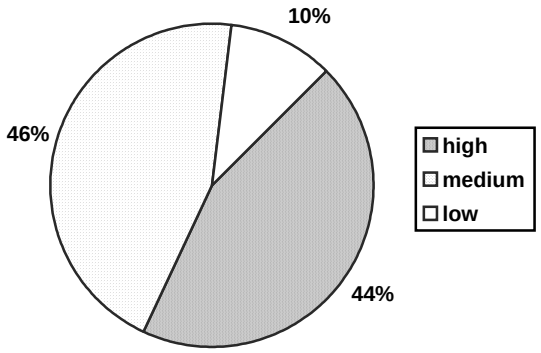


Figure 1. Categorized variability with outliers

Figure 2 presents the same data with outliers removed. This means that a violation that was classified at a higher level of variability due to the diagnostic level generated by only one compiler would now be classified at a lower level of variability, or the violation may even become consistent. However, even with the outliers removed, low variability violations still account for a small percentage of the total number of inconsistently handled violations (approximately 19.6%).

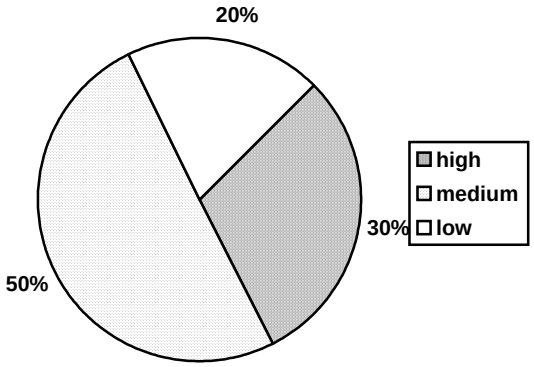


Figure 2. Categorized variability without outliers

PRODUCT AND VERSION ANALYSIS

The most common conception of portability involves the movement of code from one machine to another or from one brand of compiler to another. However, simply upgrading to a new version of the same compiler can involve the same effort as porting code between different compilers. For example, different versions of gcc change the categorization of some violations. Tables 6 and 7 illustrate this point. Table 6 counts the number of times a violation generated a warning on the given compiler (row) but resulted in an error on another (column). For instance, two violations that gcc 2.4.5 treated as warnings became errors when compiled by gcc 2.6.3. Also, one violation was an error on gcc 2.4.5 but is considered a warning in subsequent versions.

Compiler	gcc2.4.5	gcc2.6.3	gcc2.7.2
gcc2.4.5	0	2	2
gcc2.6.3	1	0	0
gcc2.7.2	1	0	0

Table 6. gcc - warning to error

Table 7 presents similar information but counts the number of times that a violation generates no diagnostic whatsoever on one version of the compiler but results in a warning on another. From this table it can be seen that there is one case where gcc 2.4.5 and 2.6.3 generate no diagnostic but gcc 2.7.2 generates a warning. There is also a violation for which 2.6.3 and 2.7.2 no longer generate a diagnostic.

Compiler	gcc2.4.5	gcc2.6.3	gcc2.7.2
gcc2.4.5	0	0	1
gcc2.6.3	1	0	1
gcc2.7.2	1	0	0

Table 7. gcc - nothing to warning

Note that these tables can be read in two ways. Reading the warning-to-error tables in row major order counts the number of times a warning on one version of the compiler becomes an error on the other. Reading the table in column major order provides counts of the number of times an error on one version of the compiler becomes a warning on another (indicated by the corresponding row). Reading the nothing-to-warning tables in row major order counts the number of times a piece of code which generates no diagnostic message on one version of the compiler generates a warning on the other. Reading the

table in column major order provides counts of the number of times a warning on one version of the compiler generates no diagnostic on another (indicated by the corresponding row). Also notice the zeroes on the diagonal since a compiler compared to itself does not generate any differences. This is not actually a rule, since the same version number of the gcc compiler can exist on different machines and operating systems, so it is possible that two installations of the same compiler could produce different diagnostic results.

The classification of violations also varies on the same machine architecture. Table 8 presents a set of PC-based C compilers and counts the number of times warnings on one compiler result in errors on another. This table indicates that of the PC compilers tested, Microsoft C most frequently generated warnings for violations that other compilers considered to be errors. If a system compiled with warnings but generated an executable under Microsoft C, it is likely that the same source code will fail to compile at all on another PC-based C compiler.

Compiler	bcc4.02	bcc5.01	msc7.0	msc10.0	zortech
bcc4.02	0	0	3	3	9
bcc5.01	0	0	3	3	9
msc7.0	10	10	0	0	17
msc10.0	12	12	2	0	19
zortech	1	1	0	0	0

Table 8. PC compilers - warning to error

Table 9 counts the number of times that a violation generates no diagnostic on one PC-based C compiler but raises an error on another.

Compiler	bcc4.02	bcc5.01	msc7.0	msc10.0	zortech
bcc4.02	0	1	6	4	8
bcc5.01	0	0	6	3	8
msc7.0	8	9	0	3	4
msc10.0	8	8	5	0	7
zortech	24	25	20	21	0

Table 9. PC compilers - nothing to error

In this table, the compiler most likely to not generate a diagnostic is Zortech. This result, and the amount of inconsistency evident in the table, is one of the reasons that the outdated version of the Zortech compiler was questioned and its inconsistent results were classified as outliers. Examining the rest of the table shows that the Microsoft C compiler most frequently fails to issue a diagnostic on code that other compilers consider erroneous. Porting code segments with these violations from Microsoft C to another PC-based compiler can cause a system that previously compiled without warning or error to completely fail to compile.

Table 10 counts the number of times a violation compiled on one compiler is not diagnosed but generates a warning on another compiler. These violations are examples of low variability violations.

Compiler	bcc4.02	bcc5.01	msc7.0	msc10.0	zortech
bcc4.02	0	0	4	3	0
bcc5.01	0	0	4	3	0
msc7.0	10	10	0	0	1
msc10.0	10	10	1	0	1
zortech	12	12	5	4	0

Table 10. PC compilers - nothing to warning

Two compilers for the OS/2 operating system were also included in the study. The results generated by these compilers concur with previous evidence that changing compilers on the same machine, with the same operating system, can result in different results, even when both compilers are instructed to compile in ANSI-compliant mode. Table 11 counts the number of differences that were identified when testing the two OS/2 compilers.

	Warning to Error		Nothing to Error		Nothing to Warning	
Compiler	bcc2.0	watcom	bcc2.0	watcom	bcc2.0	watcom
bcc2.0	0	4	0	2	0	6
watcom	1	0	7	0	6	0

Table 11. OS/2 compiler message variants

Table 12 compares the two most commonly used C compilers on the HP-UX 10.10 operating system. Keep in mind that the initial violation set was generated from gcc and therefore slants the results in favor of gcc generating better diagnostics. However, results that indicate the number of times gcc produces a warning while cc generates an error are of interest, since they still represent different handling of the same code by the two compilers.

	Warning to Error		Nothing to Error		Nothing to Warning	
Compiler	gcc2.7.2	cc	gcc2.7.2	cc	gcc2.7.2	cc
gcc2.7.2	0	29	0	0	0	0
cc	7	0	0	0	12	0

Table 12. HP-UX 10.10 compiler message variants

Table 13 gives the results from additional HP-UX compilers that were tested under version 9.05 of that operating system. It can be ascertained from these tables that the CC compiler classifies violations far differently than the other three compilers. Since CC is a pure C++ compiler and its response to the violations was so different, when a violation was classified as being of higher variability only because of the value generated by CC, that violation was considered to be an outlier. However, the results produced are quite interesting in that they indicate some discrepancy in the way C++ handles C code. The ic960 compiler is actually a derivative of the gcc compiler (one error message actually has the string “gcc” in it) but it has been modified enough so that its performance is not identical to gcc.

Compiler	Warning to Error				Nothing to Error				Nothing to Warning			
	gcc2.4.5	cc	ic960	CC	gcc2.4.5	cc	ic960	CC	gcc2.4.5	cc	ic960	CC
gcc2.4.5	0	31	5	41	0	0	0	2	0	0	0	0
cc	6	0	7	19	3	0	2	9	11	0	4	2
ic960	2	18	0	26	3	13	0	14	22	4	0	3
CC	3	4	3	0	11	16	7	0	25	13	17	0

Table 13. HP-UX 9.05 compiler message variants

Different versions of compilers from the same manufacturer also handle violations differently. Table 14 shows the number of times that a violation raised no diagnostic from one Microsoft C compiler but produced an error on a different version. These results indicate that problems may arise when compiling code that once produced no diagnostic on a newer version of your compiler.

Compiler	msc7.0	msc10.0
msc7.0	0	3
msc10.0	5	0

Table 14. Microsoft C/C++ - nothing to error

Changes to the way a compiler handles violations are not isolated to Microsoft but impact every compiler for which we had multiple versions available. Table 15 provides the counts for the Borland C++ compiler. All three versions of these compilers were on different operating systems. Version 4.02 was tested on a 486 PC with DOS 6.22 and Windows 3.1, while version 5.01 was tested on a Pentium PC with Windows 95, and version 2.0 was tested on a 486 PC with OS/2 Warp 3.0. Although the number of inconsistencies is not high, having just one of these violations in your code can cause a software system to fail to port without some modification, even if you have cleaned up all of the warning messages your compiler produced.

Compiler	bcc4.02	bcc5.01	bcc2.0
bcc4.02	0	1	1
bcc5.01	0	0	0
bcc2.0	2	2	0

Table 15. Borland C++ - nothing to error

CONCLUSIONS

Table 16 summarizes the number of gcc diagnostic messages that were identified, the number of violations that we wrote code samples for and the number and classification of inconsistently handled violations. The first column of numbers provides these counts with the outliers removed and the second column includes the outlier results in the counts.

Totals	without outliers	with outliers
messages	226	226
violated	175	175
consistent	78	60
inconsistent	97	115
high variablility	29	51
medium variability	49	52
low variability	19	12
unviolated	51	51

Table 16. Totals without and with outliers

For our set of code samples, violations of the ANSI C standard were diagnosed inconsistently more often than they were consistently. This indicates that software engineers whose ANSI-compliant C code compiles and runs in one hardware/software environment cannot expect the same code to compile and run in a different hardware/software environment without modification, even if the new compiler is ANSI C-compliant. Not only that, but the engineer can also not count on violations to be diagnosed consistently by different compilers. The patterns encountered in this research, however, have indicated certain compilers to be especially wary of, most notably Zortech and CC. The complete compiler diagnostic matrix of violations, only parts of which are reproduced here, could provide a useful mechanism for estimating portability effort. This use spills over into maintenance since maintenance often entails moving software systems to a newer version of the original compiler, and we have seen that this activity is also not without dangers. The matrix also shows conclusively that the ANSI C standard, while increasing portability over K&R C, does not cover all contingencies.

There are some limitations to this study. The first and most important is the use of gcc to obtain our initial violation set. Although in testing other compilers we did find some violations that gcc did not diagnose, the vast majority of the cases were things for which gcc did generate a diagnostic. For that reason the results presented here should not be taken to indicate that gcc produces better diagnostics. Also, we have no idea of how many violations there are that gcc does not catch that other compilers do identify. What can be determined from the results provided here is the fact that there are differences in the ways that compilers classify violations and these differences can have a tremendous impact on the portability of a software product. These differences even occur when writing ANSI C code and running the compiler in ANSI mode.

Another issue related to this study is the set of compiler options that were used when each compiler was exercised. These options can influence the number and kind of diagnostic messages that are produced. Efforts were made to keep all of the compilers equally strict, but the possibility exists that some of the compilers that failed to generate a warning during our tests may have generated a warning had different options been specified. However, this is not a significant problem. What would have changed had more strict diagnostic options been used is the number of low variability violations (which would have become consistent) and the number of high variability violations (which would have become medium variability). Stricter diagnostic options only affect which warnings are produced but should not impact on what is defined as a warning and what is defined as an error. Since the low variability category does not account for many violations, a reduction in this set does not change the overall impact of this study. The fact remains that the classification of warnings and errors by different compilers is somewhat arbitrary and can hinder portability.

REFERENCES

[ANSI/ISO90] *American National Standard for Programming Languages—C*, American National Standards Institute, NY, 1990.

[Jaeschke89] R. Jaeschke, *Portability and the 'C' Language*, Hayden Books, Indianapolis, Indiana, 1989.

[Pearse96] T. Pearse & P. Oman, "Investigations into ANSI C Source Code Portability Based on Experiences Porting Laserjet Firmware," Pacific Northwest Software Quality Conference 1996 Proceedings (Portland, OR, Oct. 1996).

[Rabinowitz90] H. Rabinowitz and C. Schaap, *Portable C*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Ryan92] R. Ryan, "A Moving Target," *Byte*, January 1992, pp. 159-174.

APPENDIX. CODE SAMPLES

High Variability Violation Samples

1011. bit-field X type invalid in ANSI C

```
struct sample {
  char b : 2;
};
```

1028. structure has no members

```
struct def { };
```

1037. comma at end of enumerator list

```
enum cat { q, w, e, };
```

1041. \x used with no following hex digits

```
const char x = '\x';
```

1043. ANSI C forbids an empty source file

1070. char-array X initialized from wide string

```
char array[] = L"Rika";
```

2061. union has no members

```
union empty{  };
```

2086. ANSI C forbids empty initializer braces

```
int xarray[5] = { };
```

Medium Variability Violation Samples

1032. width of g exceeds its type

```
struct abc {
   int g : 50;
   int j : 13;
};
```

1038. character constant too long

```
const int x = 'Am I really a string?';
```

1045. comparison between pointer and integer

```
int *p;
int i;
if (p < i);
```

1047. ANSI C forbids use of cast expressions as lvalues

```
int *p;
(float *)p = 3.5;
```

1080. invalid operands to binary -

```
int *x, *z;
float *y;
z = x - y;
```

2021. declaration of X shadows a parameter

```
int add_two(int good1, int good2);

int add_two(int good1, int good2)
{
  char good1 = 'a';

  return (good1 + good2);
}
```

2062. unnamed struct/union that defines no instances

```
struct {};
```

3042. return with a value in function returning void

```
void proc1();

void proc1()
{
  return 2;
}
```

Low Variability Violation Samples

1039. unknown escape sequence X

```
const char x = '\Q';
```

1059. multi-character character constant

```
char x = 'ab';
```

2010. X was declared implicitly extern and later static

```
foo (1,2);

static int foo(int i, int k)
{
}
```

3036. left shift count >= width of type

```
y << 99;
```

3037. left shift count is negative

```
y << -5;
```

3038. right shift count >= width of type

```
y >> 99;
```

3039. right shift count is negative

```
y >> -5;
```

4019. parameter has incomplete type

```
int dummy1(int good2, union m);
```

4037. ANSI C does not allow extra ; outside of a function

```
;

main()
{
  printf("nothing");
}
```

Fifteenth Annual Pacific Northwest Software Quality Conference
Portland, Oregon October 27-29, 1997

Stochastic Testing With an Unknown Operational Profile

Jarrett Rosenberg

Sun Microsystems
2550 Garcia Avenue, MPK17-307
Mountain View, CA 94043
Jarrett.Rosenberg@Sun.COM
Jarrett.Rosenberg@ACM.ORG

Abstract

Use of a well-defined operational profile is an essential aspect of stochastic testing of software, indeed, of any reliability assessment process. When, as is frequently the case, such a profile is not available, it is often assumed that stochastic testing therefore can not be done. In fact, stochastic testing can still be done in those circumstances; the absence of a known operational profile simply results in a quantifiable expenditure of extra testing effort to ensure the operational reliability of the product.

1.0 Introduction

Reliability is the probability of correct functioning in a given time interval, under a given set of conditions. That "set of conditions" is the *operational profile*. For simple hardware components with their few and repetitive actions, the focus is on environmental factors such as temperature and vibration. For complex systems, and especially software, the focus is on the variety of inputs and actions involved in the correct functioning of the system.

A well-defined operational profile is an essential part of the design and test of any system. It is especially critical in stochastic testing, since the statistical methods which make stochastic testing so powerful rely on the fact that the testing profile is representative of the actual operational profile (see Rosenberg, 1996).

It is therefore generally assumed that if the operational profile is not known, then there is no point in doing stochastic testing, because the results may not generalize to actual usage. In fact, there is a simple method for adjusting for lack of

information about the actual operational profile, a method which estimates upper and lower limits of the actual reliability, and which allows testers to produce a system with an assured minimum level of reliability, whatever the actual operational usage turns out to be.

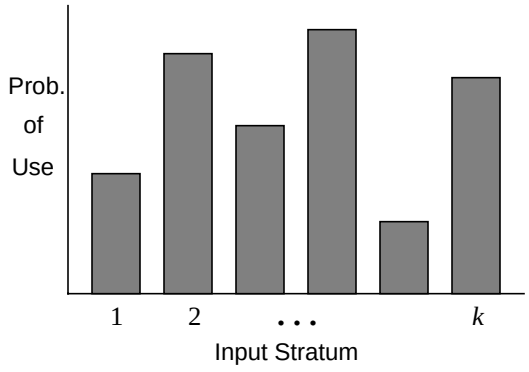
2.0 The Operational Profile and the Testing Profile

An *operational profile* defines the actual (or strongly anticipated) usage of a system in field operation. A *testing profile* defines the mix of test inputs used in a stochastic testing lab. Ideally, the testing profile for a system is identical to (more strictly, a random sample from) the system's operational profile; this is what allows us to believe that our testing results are relevant to actual usage.

In its most general form, a profile consists of two parts (see Figure 1):

- A partitioning of the input test space into *k* strata, based on some criteria (typical ones are system functions or user operations). This stratification can be fine- or coarse-grained, user-oriented or implementation-oriented, or whatever seems most appropriate; for our purposes here it doesn't matter (see Musa *et al*., 1996, for an excellent discussion).
- A probability distribution defined over that stratification, assigning to each stratum its likelihood of occurrence. This distribution is virtually always non-uniform.

FIGURE 1. The General Form of An Operational or Testing Profile



The non-uniformity of an operational profile is its central feature, and the source of many problems. In particular, its modal (most frequent) stratum must be the one that is most tested.[1] In the absence of information about the probability distribution of the operational profile, the worse case is that the modal stratum is tested least.

1. Recall that stochastic testing is a necessary but not sufficient aspect of software quality assurance, since by definition low-usage components will be lightly exercised. Safety-critical components (which are often low in expected usage) must be exhaustively tested by non-stochastic methods.

How can this misallocation of testing effort (and resulting reliability disaster) be avoided?

3.0 Adjusting for an Unknown Operational Profile

An unknown operational profile is one in which the probability distribution over the input strata is not known.[2] Since it is possible in that case for any testing profile to be the exact opposite of the actual operational profile, an obvious idea is to conduct testing under several very different profiles. A little thought reveals that such an approach tends in the limit to testing with a uniform profile, suggesting that that is where we should start.[3] If we carry out testing with a uniform profile, then we protect ourselves against the worst possible case, *viz*., that all usage takes place in input stratum *i*, while we have done no testing whatsoever there. The worst case now becomes: all usage takes place in input stratum *i*, while we have spent only 1/*k* of our testing effort there. Since it's unlikely that *all* usage would actually be concentrated in one stratum (80% is a more likely maximum), we will rarely encounter this worst case, but it is very useful to consider as a limit case.

To quantify this situation, we can consider each of the *k* input strata to have its own failure intensity[4], λ_i ; the overall failure intensity of the system will then be their average Λ. Let us rank the strata in terms of their failure intensities, and call the lowest failure intensity $\lambda_{(1)}$, and the highest $\lambda_{(k)}$. In the worst case, all usage is in the stratum with the highest failure intensity: the system's failure rate will be then be $\lambda_{(k)}$. In the best case, actual usage will be uniformly distributed (just like our testing profile), and so the actual failure intensity will be Λ.[5] The extent of our exposure from not knowing the actual operational profile can thus be expressed as the ratio of these two cases:

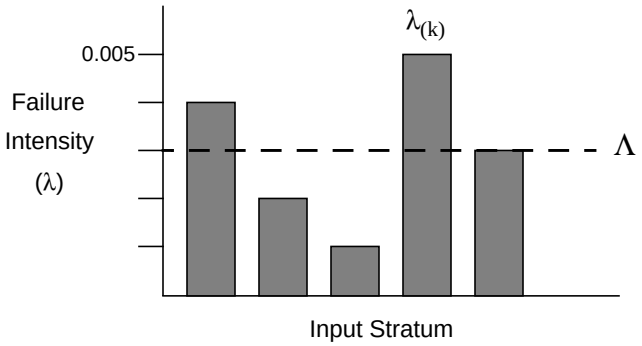
$$\text{exposure} = \lambda_{(k)}/\Lambda$$

The *a priori* probability of this worst case scenario is 1/*k*, but in practice it will be less. An example is shown in Figure 2: if the failure intensities for five strata are 0.004, 0.002, 0.001, 0.005, and 0.003, then the highest failure intensity is 0.005, the overall failure intensity (under a uniform distribution) is 0.003, and the risk (with probability ≤ 0.20) is that the failure intensity in actual usage under the worst case scenario would be 0.005/0.003 = 1.67 times higher than the estimates we obtain from our testing with a uniform profile (namely, Λ).

---

2. It's hard to imagine any meaningful testing program if the strata themselves were unknown.
3. In fact, early testing needs to be done with a uniform profile to ensure that all components are at some minimal level of reliability before serious reliability assessment can begin. There is no point in doing sophisticated reliability assessments until failures rates are within one or two orders of magnitude of their target levels.
4. The failure intensity is the expected failures per unit time, i.e., the reciprocal of the Mean Time Between Failures (MTBF).
5. The actual best case is that all the usage is concentrated in the stratum with the lowest failure intensity, $\lambda_{(1)}$, but that's being a bit *too* optimistic!

FIGURE 2. An Example of Different Failure Intensities for Different Profile Strata



It follows from this analysis that if we can reduce the size of this ratio, we reduce the severity of the worst-case scenario compared to our overall estimates using the uniform profile. Since the *k* strata have varying failure intensities, the most efficient solution is to reduce this variability, i.e., make them all equally low (i.e., equal to $\lambda_{(1)}$), at which point our exposure becomes fixed at $\Lambda=\lambda_{(1)}$.[6] Once the individual stratum failure intensities are equal, further testing then reduces Λ to the desired criterion. Note that driving the different failure intensities to equality requires a differential allocation of testing effort: those strata with higher intensities require more testing.

Once the exposure is fixed, we can then quantify the trade-off of how much testing is enough to meet our reliability criterion *R*. We can test under our uniform profile until $\Lambda \leq R$ (to within our pre-established level of confidence), but that exposes us to the possibility that the actual operation profile is non-uniform, and that our established failure rate is still too high for the high-usage components of the system (whichever ones they turn out to be). The only solution for this problem is to make *all* components of the system have reliability equal to *R*: that is the price to be paid for testing without a known operational profile.

4.0 An Example

As an example, consider an application whose target reliability is a failure intensity of 0.001, i.e., a MTBF of 1000 hours. For simplicity, let us consider a case where there are

6. We may not, in fact, have to reduce all the strata's failure intensities to that of the lowest one if we are willing to accept a failure intensity higher than that.

only five strata in the profile. After initial testing, we find that they have the following failure intensities and decay rates following the Musa-Okumoto model:[7]

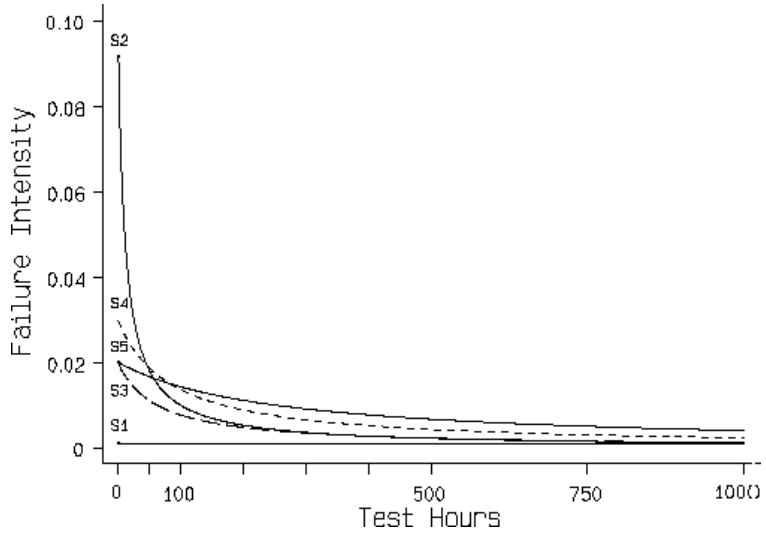
TABLE 1. A Simple Example of a Profile with Five Strata

Stratum	Initial Failure Intensity (λ_0)	Decay Rate (θ)	Failure Intensity Criterion[a]	Hours to Criterion (t)
1	0.001	0.01	0.001	0
3	0.020	0.80	0.001	1100
5	0.020	0.20	0.001	1188
4	0.030	0.40	0.001	2417
2	0.100	0.90	0.001	4750

a. The failure intensity of the stratum when the overall failure intensity, R, is at 0.001.

Figure 3 shows how the failure intensity of each stratum would decline over a testing period of 1000 test hours.

FIGURE 3. Failure Intensity of Each Stratum as a Function of Testing Time



The failure intensities vary considerably: Stratum 1 is already at our target level, which is fortunate, because its very low decay rate means it will take a huge

7. In the Musa-Okumoto model, failures follow a non-homogenous Poisson process where the failure intensity λ at time t, $\lambda(t) = \lambda_0/(\lambda_0\theta t + 1)$, where $\lambda(t)$ is the failure intensity at time t, λ_0 is the initial failure intensity, and θ is the decay rate of the failure intensity (i.e., the rate at which reliability grows as defects are removed). From this equation, time t to a given failure intensity R is given by $t = (\lambda_0 - R)/(R\lambda_0\theta)$.

amount of testing to increase its reliability. Stratum 2, on the other hand, has a very high failure intensity, which will make it the main factor in the application's reliability if it is used much. Fortunately, it has a high decay rate, so most of the bugs will be shaken out fairly quickly. The other three strata are similar in initial failure intensity, but their differing decay rates will affect how much testing time is required for each to meet the target.

At any instant in time, the application's failure intensity is that of the superimposed Poisson process formed by the sum of each stratum's failure intensity, weighted by its probability of usage. In the absence of a known operational profile, the usage weights are equal, and the application's failure intensity at the outset in this example is thus

$$0.20(0.001) + 0.20(0.100) + 0.20(0.020) + 0.20(0.030) + 0.20(0.020) = 0.032.$$

A simple calculation shows that the application will reach its goal of an overall failure intensity of 0.001 after a total of 9460 test hours

Since each stratum will be used equally, the simplest way to apportion the total reliability among the strata is to do so equally; thus each of them should have a failure intensity of 0.001. This approach also avoids the exposure we would have if the overall failure intensity were 0.001, but some strata were above and some below that level. If actual usage were concentrated in the strata with higher failure intensities, then the actual failure rate in usage would be higher than the nominal level. With equal allocation at the 0.001 level, this cannot happen.

Although the failure intensity is allocated equally among strata, the test hours are not, as shown in Table 1. This is the optimal allocation of testing resources in this case.

Suppose that just before we started our program of testing, we obtained reliable information about the operational profile; how much effort would that save us? That depends on

- how different the distribution of that operational profile was from our uniform testing profile,
- what the initial failure intensity of the most heavily used strata were,
- what the decay rate of the most heavily used strata were

For example, suppose that the operational profile in this case were that in Table 2:

TABLE 2. An Operational Profile for the Example

Stratum	Initial Failure Intensity (λ_0)	Decay Rate (θ)	Proportion of Usage	Failure Intensity Criterion[a]	Hours to Criterion (t)
1	0.001	0.01	0.10	0.0010	0
3	0.020	0.80	0.20	0.0006	484
5	0.020	0.20	0.15	0.0020	374
4	0.030	0.40	0.35	0.0010	858
2	0.100	0.90	0.20	0.0005	484

a. The failure intensity of the stratum when the overall failure intensity, R, is at 0.001.

Then the initial overall failure intensity would be

$$0.10(0.001) + 0.20(0.10) + 0.20(0.02) + 0.35\ (0.03) + 015(0.02) = 0.0375.$$

After 2200 hours of testing (as shown in the table) we would reach our overall goal of 0.001, a saving of 7260 hours—over 75%! This is the value of having an operational profile. One can use this fact to get a rough estimate of whether it makes more economic sense to spend the time and money to obtain an accurate operational profile (typically 2-20 months), or to do the extra testing required in the absence of such a profile.

5.0 Testing With A Partially Known Operational Profile

In some cases, the operational profile is not entirely unknown, but is only partially known. In some cases, the operational profile for a previous version of the software is known, but that for newly added functionality is unknown. This case is easily handled by the method outlined above.

More commonly, a partially known operational profile is one in which only the relative, rather than the absolute, frequency of use is known for each stratum. While there is a method that is useful in this case, it is more complicated than testing with a completely unknown profile, and the risk it entails cannot be precisely quantified. Nevertheless, it may still be of use.

The procedure for testing with a partially known profile has two steps:

1. Create a partial order of the various strata in terms of their level of expected frequency of use,
2. Assign each level of relative usage an absolute probability of usage for testing purposes.

Neither of these steps can be performed with absolute assurance. However, we can take advantage of the Pareto principle (sometimes called the "80/20 Rule"), which states that most of a system's activity comes from a small number of its components. Thus we would expect only a small proportion of the strata to be in the high-use level, and the probability of usage of those strata will be one or more orders of magnitude higher than those in the lowest-use level. These constraints reduce our chance of going seriously astray.

In the first step, each of the many strata is assigned some level of relative usage in a partial order (ties allowed) of usage frequency ranks. With a large number of strata, it is infeasible to use many levels in such an ordering: even if one were capable of making numerous fine distinctions in usage frequency, the number of relative comparisons among strata gets quickly out of hand. Instead, a relatively small number of usage levels (5-10) should be used. For example, the usage levels might be defined as "very high", "high", "medium", "low", and very low".[8]

In the second step, all of the strata in a given usage level are assigned an absolute probability of usage, as is required in the testing process. This is perhaps the more problematic part of the procedure.

As an example, suppose we have a system whose strata are sorted into the usage levels given in Table 3.:

TABLE 3. A Partially Known Operational Profile

Relative Usage Level	% of Strata	Absolute Probability of Usage	Avg. Initial Failure Intensity (λ_0)	Avg. Decay Rate (θ)	Failure Intensity Criterion[a]	Hours to Criterion (t)
"Very High"	10%	.60	.002	.10	.0013	1752
"High"	10%	.20	.020	.80	.0004	584
"Medium"	20%	.10	.100	.90	.0004	292
"Low"	30%	.05	.030	.40	.0008	146
"Very Low"	30%	.05	.020	.20	.0016	146

a. The failure intensity of the stratum when the overall failure intensity, R, is at 0.001.

Given this apportionment of strata to levels, and assignment of probabilities to those levels, the overall failure intensity, Λ, is 0.013 at the outset. Testing with this profile will reach its goal of an overall failure intensity of 0.001 after a total of 2920 hours.[9] Under a uniform profile, reaching the overall 0.001 level would require a

8. Theoretically, one could proceed recursively and further partition each level by usage, but this is unlikely to be useful in practice. However, one might divide each level into strata of high, medium, and low failure intensities, allocating more testing to the first of these.

9. Note that in this example, even though the overall failure intensity is 0.001, the most frequently used strata have a failure intensity that is slightly higher; this is because of the slow decay rate of the failure intensity: 5000 hours to go from 0.002 to 0.0013. It may not be economically feasible to lower it further.

total of 14,460 hours. The nominal worst case (with probability ≤ 0.05) would appear to be that, in actual usage, the strata in the "Very Low" usage level would receive all the usage, with the resulting exposure of 0.0016/0.001 = 1.6 times the nominal failure intensity. However, this is not true, since there are two ways of going astray with this procedure:

- *Misallocation of strata to levels*. Any number of strata could have been placed in the wrong usage level. Most frequently, a stratum will be allocated to the usage level just above or just below its true level, but the larger the number of strata, the more likely such misallocations will occur. A possible remedy is to provide an extra margin of safety by assigning the second-highest usage level the same absolute usage probability as the highest usage level.
- *Misspecification of the absolute usage probability for the levels*. This mapping, between a small scale and a large one, contains the greatest chance of error. Unfortunately, the obvious solution of further increasing the probabilities for the highest (or two highest) levels also causes the probabilities to be lowered for the other levels. Errors here compound those introduced in the first step.

Thus, while this method is extremely efficient if the assumed profile is accurate, it is very difficult to precisely quantify the risk when the profile is inaccurate. It should therefore not be used unless there is a strong degree of confidence in the profile. A comparison between the testing efforts involved in using an unknown or partially known operational profile can aid management in deciding which approach to pursue, given the level of confidence in the partially known profile.

6.0 Conclusion

It can be seen, therefore, that lack of a known operational profile does not mean that successful stochastic testing and reliability assessment cannot be carried out, it means that more effort will be needed to reach the desired reliability level, since all parts of the system must be made as reliable as the overall system, even if those parts may be rarely used.

While this method requires an extra expenditure of effort (often a considerable one), it has the advantage of quantifying that effort, making it fairly obvious to management how much more economical it is to obtain the actual operational profile, rather than compensating for that lack with extra testing.

References

Musa, J., G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin, "The operational profile" in M. Lyu, ed., *Handbook of Software Reliability Engineering*. NY: McGraw-Hill. 1996.

Rosenberg, J. "Software testing as acceptance sampling." *Proceedings of the Fourteenth Annual Pacific Northwest Software Quality Conference*. Portland, OR. Sept. 1996.

1997 Proceedings Order Form

Pacific Northwest Software Quality Conference

To order a copy of the 1997 Proceedings, please send a check in the amount of $35.00 to:

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

Name_____

Affiliate_____

Mailing
Address_____

City_____

State_____

Zip_____

Daytime
phone_____