NASA/TM-1998-208617

SOFTWARE ENGINEERING LABORATORY SERIES SEL-96-002

PROCEEDINGS OF THE TWENTY-FIRST ANNUAL SOFTWARE ENGINEERING WORKSHOP

DECEMBER 1996

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Proceedings of the Twenty-First Annual Software Engineering Workshop

December 5–6, 1996

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

Page intentionally left blank

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Flight Dynamics Systems Branch

The University of Maryland, Department of Computer Science

Computer Sciences Corporation, Development and Systems Engineering organization

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effects of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Documents from the Software Engineering Laboratory Series can be obtained via the SEL homepage at:

http://fdd.gsfc.nasa.gov/seltext.html

or by writing to:

Flight Dynamics Systems Branch
Code 551
Goddard Space Flight Center
Greenbelt, Maryland 20771

The views and findings expressed herein are those of the authors and presenters and do not necessarily represent the views, estimates, or policies of the SEL. All material herein is reprinted as submitted by authors and presenters, who are solely responsible for compliance with any relevant copyright, patent, or other proprietary restrictions.

CONTENTS

Materials for each session include the viewgraphs presented at the workshop and a supporting paper submitted for inclusion in these *Proceedings*.

Page

CONTENTS (cont'd)

Omit
this
page

Session 1: The Software Engineering Laboratory

The Improvement Cycle: Analyzing Our Experience
R. Pajerski, NASA/Goddard, and S. Waligora, Computer Sciences Corporation

Evolving the Reuse Process
S. Condon, Computer Sciences Corporation, S. Kraft, NASA/Goddard , V. Basili,
J. Kontio, C. Seaman, and Y. Kim, University of Maryland

Studies on Reading Techniques
V. Basili, G. Caldiera, F. Shull, and F. Lanubile, University of Maryland

51-61

415644

360671

The Improvement Cycle: Analyzing Our Experience

p. 24

Rose Pajerski
NASA, Goddard Space Flight Center
Flight Dynamics Division
Greenbelt, MD 20771
(301) 286-3010
rose.pajesrki@gsfc.nasa.gov

Sharon Waligora
Computer Sciences Corporation
4061 Powder Mill Rd.
Calverton, MD 20705
(301) 572-3751
swaligor@csc.com

Abstract

NASA's Software Engineering Laboratory (SEL), one of the earliest pioneers in the areas of software process improvement and measurement, has had a significant impact on the software business at NASA Goddard. At the heart of the SEL's improvement program is a belief that software products can be improved by optimizing the software engineering process used to develop them and a long-term improvement strategy that facilitates small incremental improvements that accumulate into significant gains. As a result of its efforts, the SEL has incrementally reduced development costs by 60%, decreased error rates by 85%, and reduced cycle time by 25%. In this paper, we analyze the SEL's experiences on three major improvement initiatives to better understand the cyclic nature of the improvement process and to understand why some improvements take much longer than others.

Background

Since 1976, the Software Engineering Laboratory (SEL) has been dedicated to understanding and improving the way in which one NASA organization, the Flight Dynamics Division (FDD) at Goddard Space Flight Center, develops, maintains, and manages complex flight dynamics systems. It has done this by developing and refining a continual process improvement approach that allows an organization such as the FDD to fine tune its process for its particular domain. Experimental software engineering and measurement play a significant role in this approach.

The SEL is a partnership of NASA Goddard's FDD, its major software contractor, Computer Sciences Corporation (CSC), and the University of Maryland's (UM) Department of Computer Science. The FDD primarily builds software systems that provide ground-based flight dynamics support for scientific satellites. They fall into two sets: ground systems and simulators. Ground systems are midsize systems that average around 250 thousand source lines of code (KSLOC). Ground system development projects typically last approximately 2 years. Most of the systems have been built in FORTRAN on mainframes, but recent projects contain subsystems written in C and C++ on workstations. The simulators are smaller systems averaging around 60 KSLOC that provide the test data for the ground systems. Simulator development lasts between 1 and 1.5 years. Most of the simulators have been built in Ada on a VAX computer. The project characteristics of these systems are shown in Table 1. The SEL is responsible for the management and continual improvement of the software engineering processes used on these FDD projects.

Table 1. Characteristics of SEL Projects

Characteristics	Applications	
	Ground Systems	Simulators
System Size	150 - 400 KSLOC	40 - 80 KSLOC
Project Duration	1.5 - 2.5 years	1 -1.5 years
Staffing (technical)	10 - 35 staff-years	1 - 7 staff-years
Language	FORTRAN, C, C++	Ada, FORTRAN
Hardware	IBM Mainframes, Workstations	VAX

The SEL process improvement approach shown in Figure 1 is based on the Quality Improvement Paradigm [Reference 1] in which process changes and new technologies are 1) selected based on a solid *understanding* of organization characteristics, needs, and business goals; 2) piloted and *assessed* using the scientific method to identify those that add value; and 3) *packaged* for broader use throughout the organization. Using this approach, the SEL has successfully established and matured its process improvement program throughout the organization.

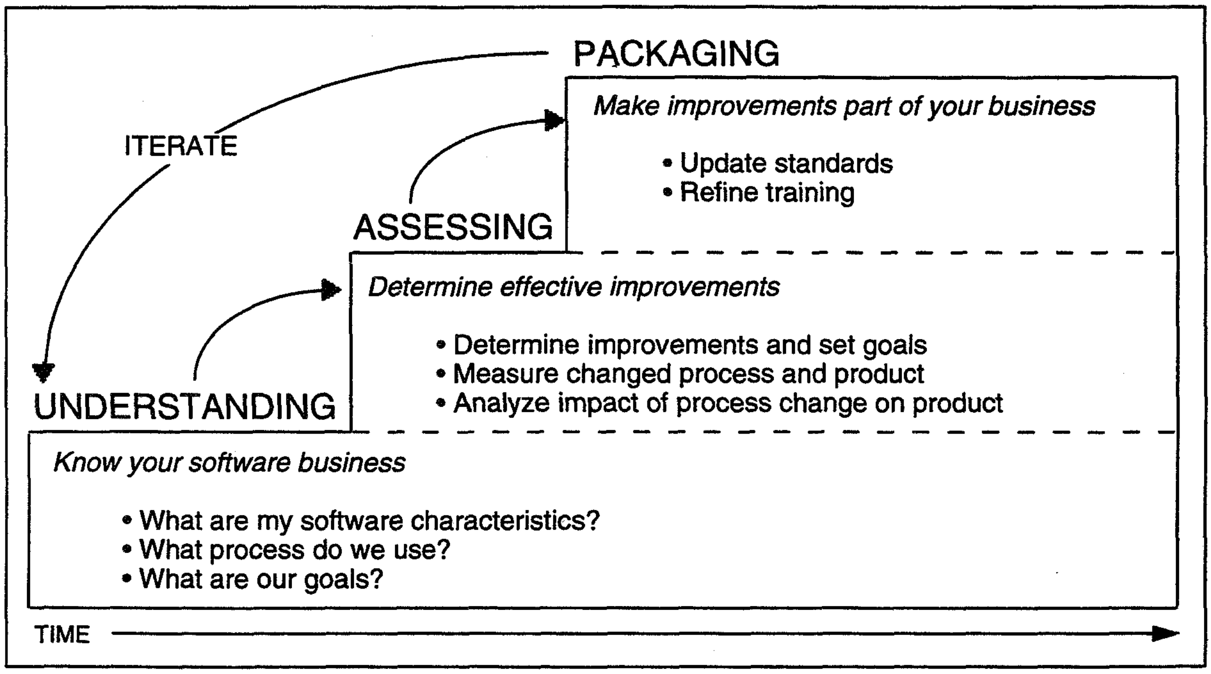

PACKAGING
Make improvements part of your business
• Update standards
• Refine training
ITERATE
ASSESSING
Determine effective improvements
• Determine improvements and set goals
• Measure changed process and product
• Analyze impact of process change on product
UNDERSTANDING
Know your software business
• What are my software characteristics?
• What process do we use?
• What are our goals?
TIME

Figure 1. SEL Process Improvement Paradigm

The SEL organization consists of three functional areas: software *developers*, software engineering *process analysts*, and *data base support* (Figure 2). The largest part of the SEL is the 150 to 200 software personnel who are responsible for the development and maintenance of over 4 million source lines of code (SLOC) that provide orbit and attitude ground support for all Goddard missions. Since the SEL was founded, software project personnel have provided software measurement data on over 130 projects. This data has been collected by data base support personnel and stored in the SEL data base for use by software project personnel and process analysts. The process analysts are responsible for defining the experiments and studies, analyzing the data, and producing reports. These reports affect such things as project standards, development procedures, and how projects are managed. The data base support staff is responsible for entering measurement data into the SEL data base, quality assuring the data, and maintaining the data base and its reports.

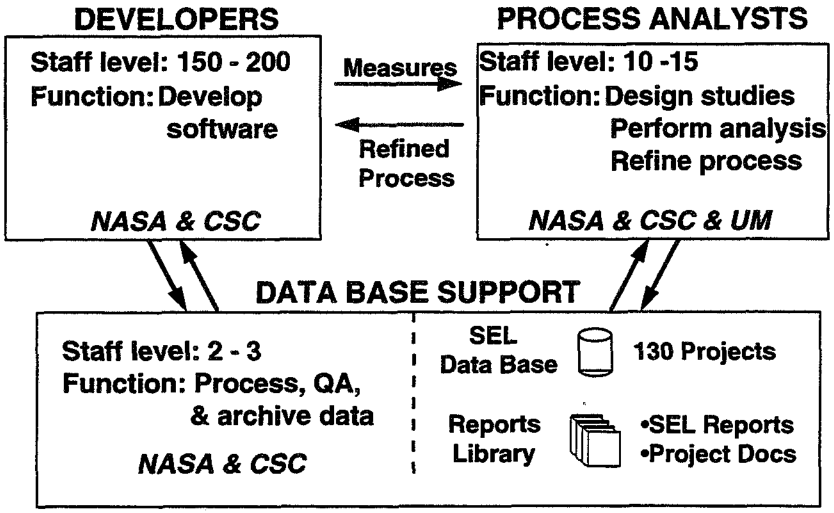


Figure 2. SEL Organizational Structure

Improvement Cycles

Although the improvement process is a never-ending endeavor, it is cyclic in nature. At the SEL, improvement cycles operate within the context of the SEL process improvement paradigm. Each improvement cycle tends to focus on a single organizational goal and only one or two process or technology changes that address that goal. Often these build on earlier experimental results. Each SEL improvement cycle has four major steps:

1. Each improvement cycle begins with setting improvement goals based on the current business needs and strategic direction of the organization. Based on a solid understanding of the problem domain (application), the development environment, and the current process and product characteristics of the organization, process analysts identify leverage areas, i.e., software process or product characteristics that could have a significant impact on the overall performance of the organization. For example, increasing software reuse would have a high probability of reducing project cost and development cycle time. Therefore, if the business goals are to reduce cost and/or cycle time, increasing reuse would be a reasonable leverage area.

2. The next step is to identify software engineering technologies (processes, methods, and/or tools) that are likely to affect the leverage area. For example, object-oriented techniques (OOT) are reported to facilitate reuse. The ultimate goal of this step is to select one technology or process change that has the greatest potential for meeting the improvement goal.

3. The third and longest step of the improvement cycle is to conduct experiments to understand the value and applicability of the new technology in the local organization. Scientific methods are used to pilot the technology on one or more real projects and observe the use and effect of the

technology on the development process, products, and project performance. Process analysts use both qualitative feedback and quantitative measurements to evaluate the value of the technology/process change. Key project measurements are compared with those from a control group (similar contemporary projects using the standard process) to assess overall value. Several experiments that successively refine the process/technology may be required before it is ready to deploy.

4. The final step in an improvement cycle is to deploy the beneficial process/technology throughout the organization. This involves integrating the process change/technology into the standard process guidebooks, providing training to project personnel, and providing ongoing process consulting support to facilitate the adoption of the new technology/process change.

Since its inception, the SEL has completed numerous improvement cycles spanning from 1 to 7 years. The amount of time it takes to complete a cycle depends on the maturity and breadth of the technology/process change. Several improvement cycles are usually active at one time; however, they involve different subsets of the organization's projects.

SEL Improvement Examples

In 1985, the SEL set two fairly common improvement goals: 1) reduce the cost of developing software systems and 2) improve the quality of delivered systems. In 1990, in response to NASA's new emphasis on launching missions more quickly, a third goal was adopted: 3) reduce the cycle time needed to develop new systems. All of these goals were addressed by leveraging different process and technology areas within the context of a unified improvement program.

The following examples illustrate the different approaches taken and results achieved within three representative SEL improvement initiatives. As shown in Table 2, each initiative used a different number of improvement cycles and a somewhat different deployment strategy to achieve the desired results. The number of improvement cycles was driven by the experiment approach and results, while the deployment strategy was selected based on a risk/benefit analysis of the process change using the experiment results.

Table 2. SEL Improvement Examples

Goal	Improvement Initiative	Cycles	Experimentation Approach	Deployment Approach
Reduce Cost	Maximize Reuse	2	Iterative learning of how to apply OO concepts; develop new reuse methods	Full use in highest payback applications (subset of projects)
Increase Quality	Leverage Human Discipline	3	Iterative refinement of existing, external testing techniques and Cleanroom Methodology	Subset of 'best' techniques across all projects
Reduce Cycle Time/Cost	Streamline Testing Process	1	Refine and consolidate local (familiar) processes	Full use across all projects

Example 1: Maximizing Reuse

To reduce costs, the SEL chose to introduce and experiment with two software engineering technologies, the Ada language and object-oriented design (OOD), that had high potential for maximizing software reuse. Experimentation began across a single class of applications, flight dynamics simulators, as the first improvement cycle focused on defining a generalized architecture based on more theoretical OO concepts. Once the developers were able to apply the architecture to their systems, the application scope expanded to include generalizing more elements of flight dynamics systems. The second group of experiments expanded the definition of 'generalized' to include reusable specifications, which has resulted in a large library of reusable flight dynamics components. Figure 3 shows the experimental focus areas and timeline for these two improvement cycles. Because the early work with OO was more conceptual, several phases of experimentation across different development projects were undertaken prior to deploying the supporting process changes.

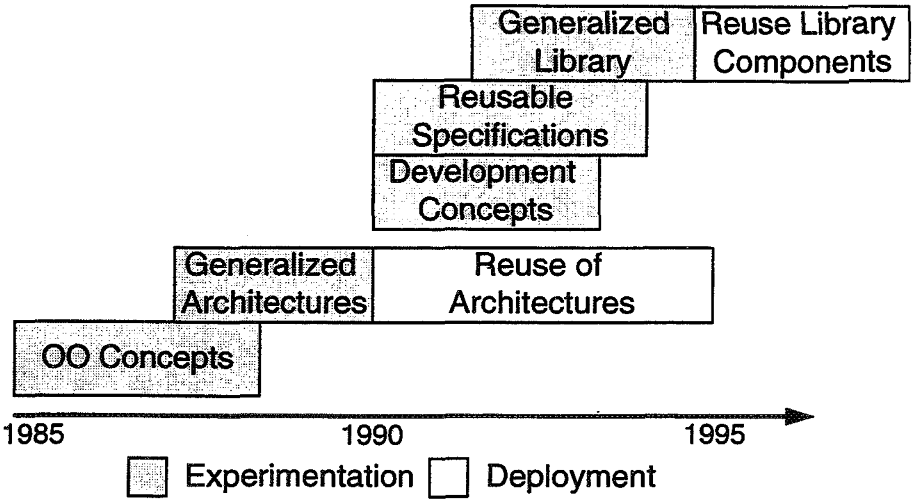


Figure 3. SEL Reuse Improvement Cycle Timeline

Within 4 years, this effort culminated in the first deployment of reusable generalized architectures that have led to a 300% increase in software reuse per system and an overall cost reduction of 55% during the next 4 years [Reference 2]. Further development of these object-oriented concepts has produced a set of reusable specifications and a corresponding component library that promises even greater improvements in 1997 systems. Figure 4 depicts the measured impact to the FDD resulting from these changes.

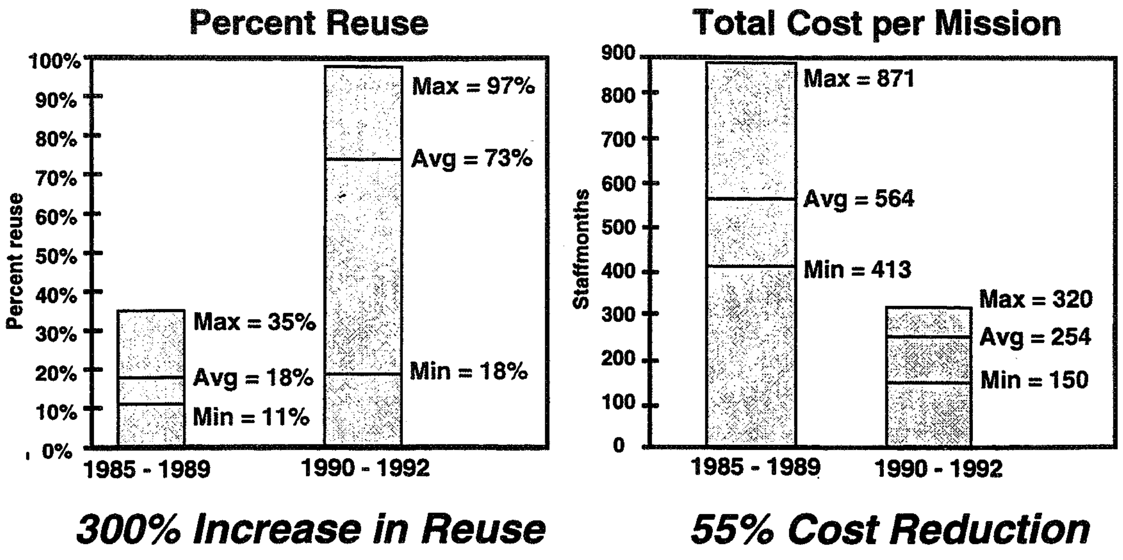


Figure 4. Results of Introducing OOD and Ada

Example 2: Leveraging Human Discipline

Early experimental results showed the positive impact on software development from leveraging the experience and perspective of the individual developer. Based on these results, the SEL chose to focus on software engineering methodologies that support human discipline to meet our quality goal [Reference 3]. The first improvement cycle, which investigated different testing techniques such as code reading and unit and functional testing, confirmed that those methods which relied on human discipline were the most effective. This led to a significant effort within the SEL to maximize the potential of human discipline by experimenting with the Cleanroom Methodology [Reference 4].

The SEL has completed two improvement cycles over four projects (two large, two small) that specifically addressed Cleanroom; the initial SEL Cleanroom project began in 1988, with the fourth and final effort completed this year. The focus of the Cleanroom Methodology is on producing software with high probability of zero defects. The key elements of the methodology include an emphasis on human discipline in the development process via code inspections and requirements classification, and a statistical testing approach based on anticipated operational usage. Development and testing teams operate independently, and all development team activities are performed without on-line testing. Analysis of the first three Cleanroom efforts had indicated greater success in applying the methodology to smaller (< 50K developed lines of code (DLOC)) in-house Goddard projects than to larger scale efforts typically staffed by joint contractor-government teams. The final Cleanroom project involved the development of a large-scale system (480K SLOC, 140K DLOC). The primary study goal was to examine it as an additional data point in the SEL's analysis of Cleanroom applicability in the organization, especially in the area of scalability.

The goal of the SEL's Cleanroom study was not to make a decision on adopting Cleanroom in its entirety within the organization, but rather to highlight those aspects of the methodology that had favorable impacts and to incorporate them into the standard approach. This approach of incremental deployment, shown in Figure 5, proved very successful in instilling these changes throughout the organization. Experimentation with Cleanroom raised the general awareness of the organization regarding quality techniques and discipline-enhancing processes. This emphasis is one of the key reasons for the FDD's steady improvement in reducing development error rates by 85% over a 15-year period, as shown in Figure 6.

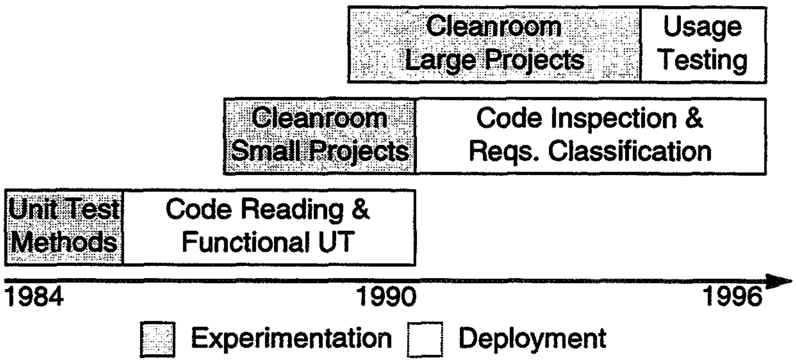


Figure 5. SEL Quality Improvement Cycle Timeline

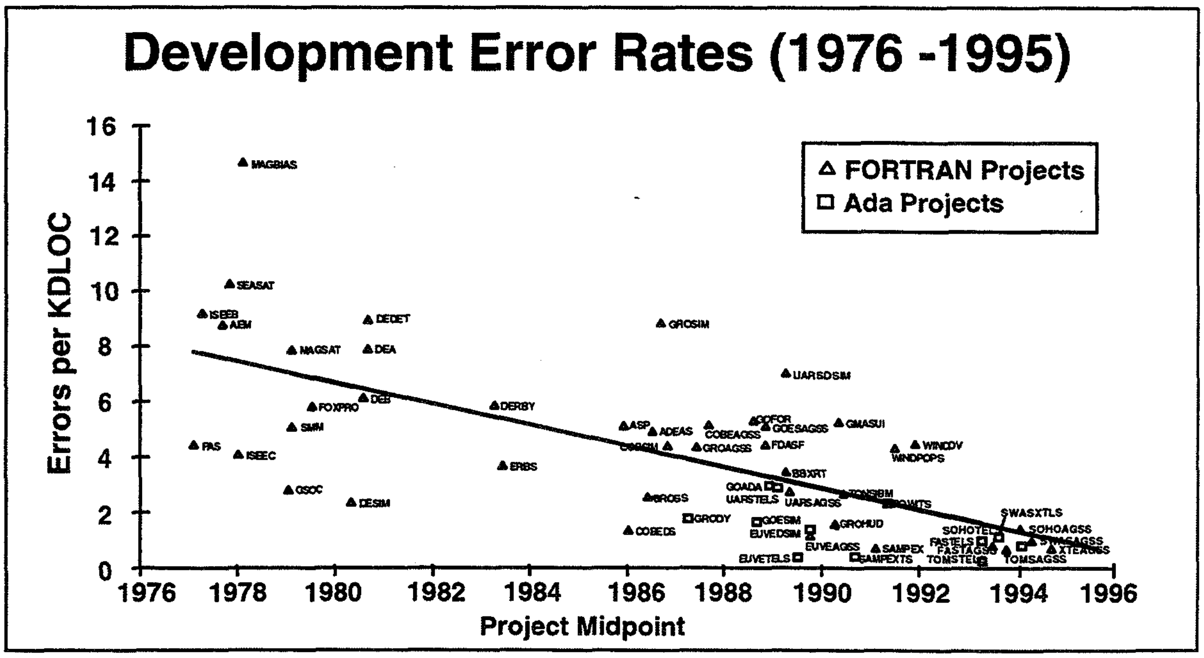


Figure 6. Quality Improvement in the SEL

Example 3: Streamlining the Testing Process

In 1992, the SEL saw the cost of system development decreasing significantly due to increasing code reuse; however, no corresponding decrease in development cycle time was occurring. In addition, although the cost associated with design and code effort had been reduced, testing costs remained virtually the same. This led to an assessment of the testing processes in use and resulted in a decision to focus testing in one group. This group, called the independent testers, effectively collapsed two separate testing phases (system and acceptance) conducted by two different groups (developers and users) into one phase (independent) performed by one group composed of experienced flight dynamics analysts and testers. This change, in both process and organization, was introduced in order to reduce the cycle time required to deliver a system, to improve the efficiency of the testing process, and to do so without sacrificing the quality of any product delivered.

Since this change was limited to one organization that was already heavily involved in defining the new testing process, the experimentation portion was brief and the risk of full deployment was judged to be low (Figure 7). Once the organizational changes were made, process changes were implemented quickly, simultaneously across all current test efforts. The resultant measurements (Figure 8) indicate that independent testing has yielded a definite shift in life-cycle effort distribution, with the testing effort being reduced from 41% to 31% of the total project effort [Reference 5]. Reductions in cycle time on the order of 5% to 20% have been verified with no loss of quality.

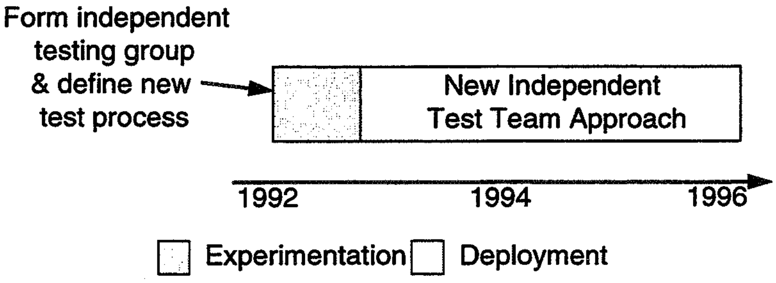

Form independent testing group & define new test process
New Independent Test Team Approach
1992
1994
1996
Experimentation
Deployment

Figure 7. SEL Independent Testing Improvement Cycle Timeline

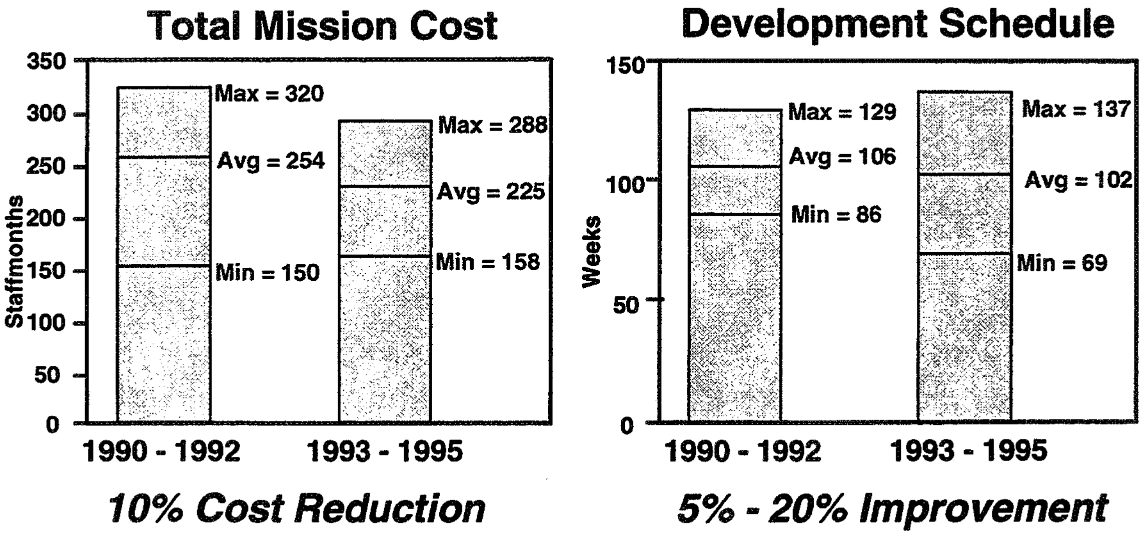


Figure 8. Results of Streamlining the System Testing Process

Measuring Overall Improvement

Each of the above initiatives resulted in measurable improvement; however, each was measured in isolation on a particular set of projects. On a long-term, continual improvement program, it is important to periodically assess how the organization is doing as a whole. To make this assessment and to update the organizational characteristics that will drive future improvement decisions, the SEL periodically computes an organizational baseline. This consists of key measurements that characterize the performance of the project organization over a specified time period and represent the organization's ability to perform similar work in the future.

We use a fairly small set of baseline measurements to evaluate improvement. They include total cost, total duration, development error rate, reuse percentage, cost per new line of code, and cost per delivered line of code. For each baseline measurement, a maximum, a minimum, and a project mean are computed for a particular time period, referred to as the baseline period. Overall improvement in each measurement is determined by comparing the means of two baseline periods, i.e., (current mean - previous mean) / previous mean.

Since 1985, the SEL has computed three baselines to measure overall improvement. Figure 9 shows when these baselines were computed in relation to the three examples discussed earlier. Notice that baselines were computed a few years after a set of improvements were deployed, allowing time for projects to use the improved process. Figures 10 and 11 show how the results of the individual initiatives combined to make significant overall improvements.

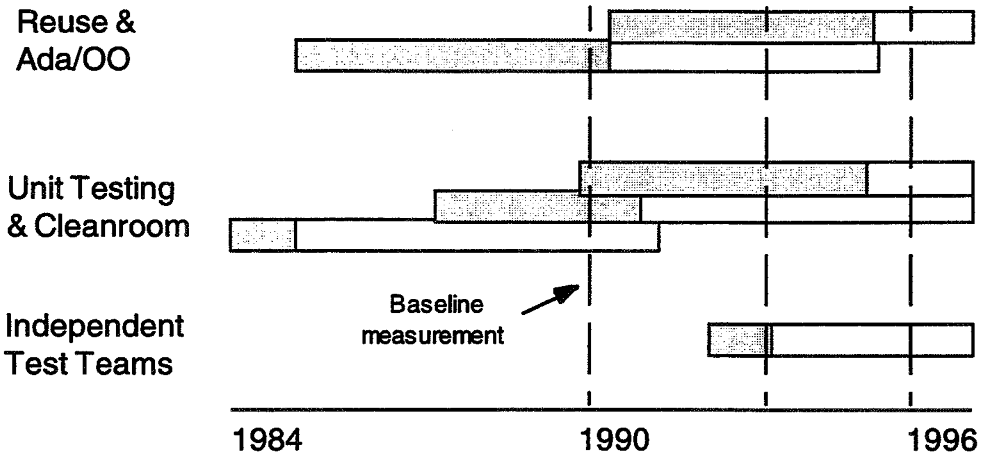


Figure 9. Improvement Cycle Timelines

The SEL's recently completed 1996 organizational baseline shows across-the-board improvement in all measurements:

- Average mission cost decreased by 15% when compared with the 1993 baseline, totaling a 60% overall reduction in mission cost since 1985 (Figure 10).
- The cost to develop a line of new code decreased nearly 35% since 1993. (There had been no previous improvement in this measure.)
- Ground system projects saw a modest 7% reduction in project cycle time, while simulators experienced a 20% reduction since 1993 (Figure 8).
- Error rates continued to drop, with a 40% reduction in development error rates since 1993. This combines with earlier improvements to total an 85% drop in development error rates over the past 10 years (Figure 6).
- After the initial 300% increase in reuse seen in the 1993 baseline, software reuse remained high, with an average of 80% on all projects; however, the minimum amount of reuse has now risen from 18% in the 1993 baseline project set to 62% in the recent project set, demonstrating a much more consistent use of reusable products in the SEL (Figure 11).

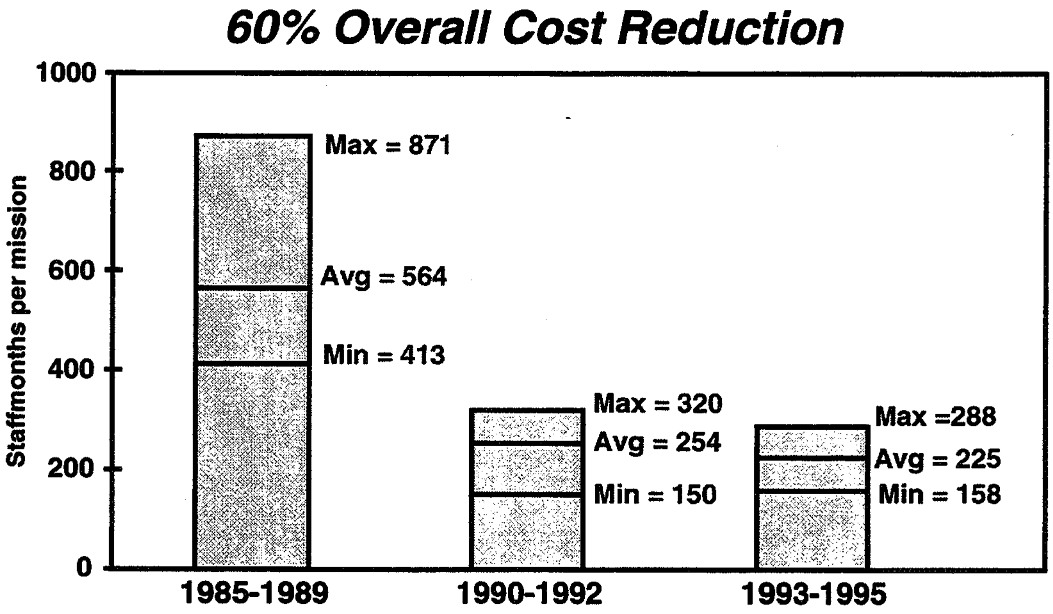


Figure 10. Overall Cost Reduction in SEL

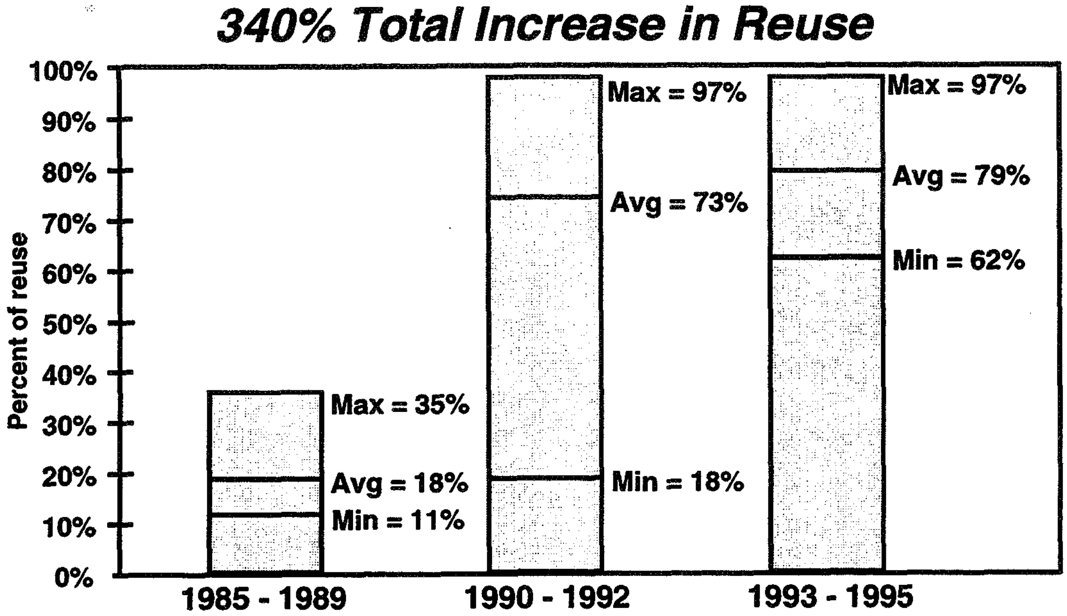


Figure 11. Overall Improvement in Reuse

Observations and Conclusions

The SEL's success with incremental process change, as opposed to leading-edge technology adoption, has led us to select the experimental approach to changing process gradually. Experimentation has allowed the beneficial changes to be deployed incrementally with low risk to ongoing projects. Deployment has been quicker for those process changes that were confined to a single phase or development activity, as with the test team process change. Following are several observations and recommendations based on our analysis of the improvement cycles discussed in this paper.

• *Focus on a single goal for each process/technology change* to provide a clear definition of the expected change and non-ambiguous measurement of its effect. There is a temptation to overload a single project with multiple changes, often in the hope that at least one will work. SEL experience suggests that this approach will not result in sustained improvement; it will only confuse the team and obscure the impact of the individual technologies.

• *Select process changes that leverage peoples' talents.* Processes that enhance human discipline and intellectual abilities provide significant improvement. Tools should be used to replace or facilitate routine tasks such as configuration and change management.

• *Allocate sufficient experimental time for tailoring and iterative application/learning of new concepts.* The SEL's experience in first developing OOD concepts followed by a generalized architecture, prior to deployment, shows the benefit of taking a little more time to develop a more usable product (the architecture) rather than deploying the more abstract concepts first.

• *Set improvement time expectations appropriately.* The more familiar the organization is with the process being changed, the faster it can be tuned and deployed and its impact realized. Existing processes can be refined and adapted more rapidly than abstract concepts; however, adapting an external (unfamiliar) process, such as Cleanroom, will take longer than refining an existing local process, such as streamlining the SEL testing process.

• *Deploy a subset of the changes as soon as the benefit is shown.* Often it is clear that certain subprocesses or techniques are very beneficial even though the entire new process/technology may not yet be proven. Early deployment allows the organization to reap its benefits as early as possible and paves the way for the rest of the method that may follow.

References

[1] Basili, V., "Quantitative Evaluation of a Software Engineering Methodology," Proceedings of the First Pan Pacific Computer Conference, Melborne, Australia, September 1985.

[2] Waligora, S., M. Stark, and J. Bailey, "The Impact of Ada and Object-Oriented Design in NASA Goddard's Flight Dynamics Division," *Proceedings of the 13th Annual Washington Ada Symposium (WAdaS96)*, July 1996.

[3] Basili, V. and R. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions of Software Engineering*, Vol. SE-13, No. 12, December 1987.

[4] Basili, V. and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994, pp. 58-66.

[5] Waligora, S. and R. Coon, "Improving the Software Testing Process in NASA's Software Engineering Laboratory," *Proceedings of the Twentieth Annual Software Engineering Workshop*, Goddard Space Flight Center, December 1995.

The Improvement Cycle: Analyzing Our Experience

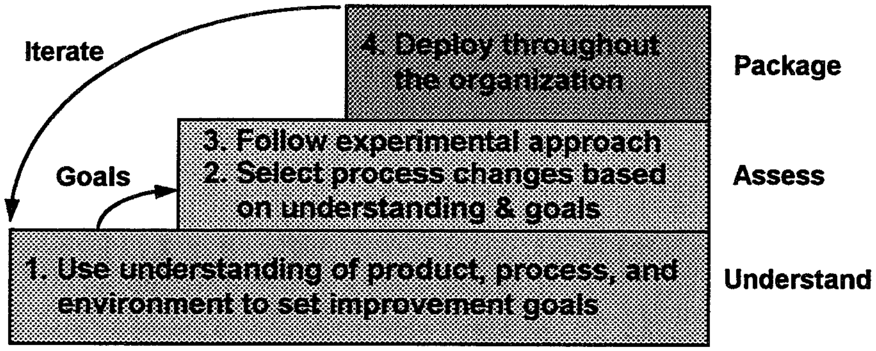
Rose Pajerski, NASA GSFC
Sharon Waligora, CSC

Presentation Outline

- What is an improvement cycle?
  - Relationship to SEL Improvement Approach
  - Improvement cycle steps
- Compare/contrast SEL examples
  - Reuse
  - Quality Techniques
  - Independent Testing
- Observations and Conclusions

What Is an Improvement Cycle?

Iterations of experimentation followed by deployment to satisfy an organizational goal



Iterate

4. Deploy throughout the organization

Package

Goals

3. Follow experimental approach
2. Select process changes based on understanding & goals

Assess

1. Use understanding of product, process, and environment to set improvement goals

Understand

SEL Improvement Paradigm

Step 1 - Use Understanding of Process and Environment

- What's inside/outside organization's control (requirements changes, deadlines)
- Current baseline measures of organizational performance (effort, schedule, errors)
- Process characteristics (work activities)
- How people spend their time

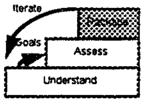
Step 2 - Select Process Change Based on Organizational Goals

Goal	Leverage Area	Experimental Focus
Decrease Cost	Maximize reuse	Ada, object oriented techniques
	Minimize rework	CASE
	Eliminate process redundancy	Combine phases
Increase Quality	Increase personal discipline	Cleanroom, personal software process
	Detect errors earlier	Testing and review methods

Step 3 - Follow Experimental Approach

- Select measures to fulfill experimental goals
- Iterate on multiple projects, using multiple techniques
  - Established methods: Pilot/Refine
  - Conceptual methods: Create/Pilot/Refine
- Involve development organization in feedback loop

Step 4 - Deploy Throughout Organization



- Document process to appropriate level
- Provide training for new element in the context of the existing process
- Reinforce use by publicizing results to development organization

Example 1 - Reuse

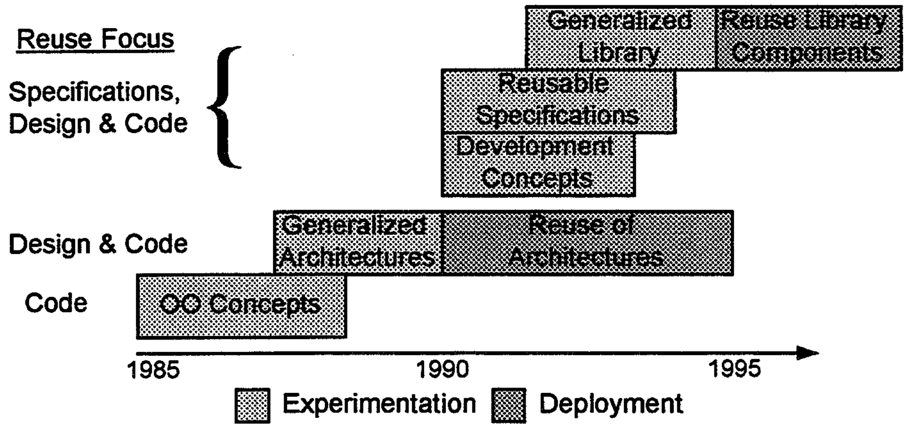
1	Improvement Goal	• Reduce cost
	Baseline Measures	• 20% code reuse per system • 564 staffmonths per mission
2	Leverage Area	• Increase software reuse
	Process or Technology	• Use Ada language • Apply object-oriented concepts
3	Expectations	• 40% code reuse per system • Reduced cost per mission
	Experiment Approach	• Iterative learning • Multiple small projects
4	Deployment	• Full use in highest payback applications • Just-in-time training by local experts

Reuse Improvement Cycles

2 major improvement cycles

Iterative learning of how to apply OO concepts

Scope: Increased from code to specifications reuse



Reuse Focus

Specifications, Design & Code

Design & Code

Code

Generalized Library

Reuse Library Components

Reusable Specifications

Development Concepts

Generalized Architectures

Reuse of Architectures

OO Concepts

1985

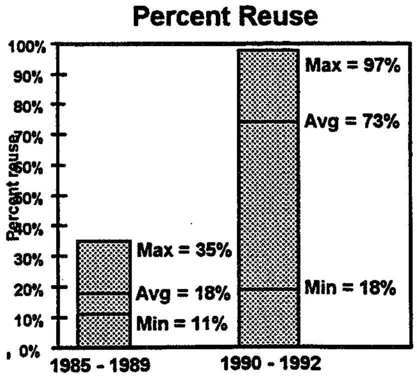
1990

1995

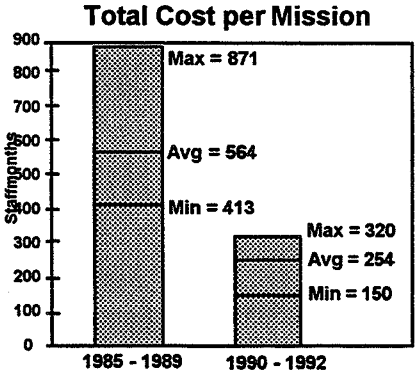
▨ Experimentation ▨ Deployment

Reuse - Results of First Improvement Cycle

Improvement exceeded expectations.



300% Increase in Reuse



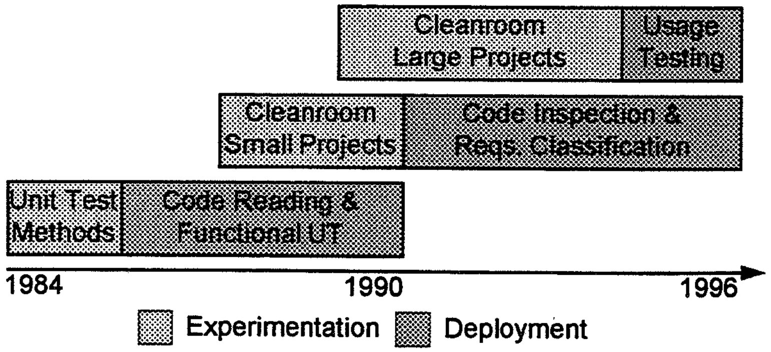
55% Cost Reduction

Example 2 - Quality Techniques

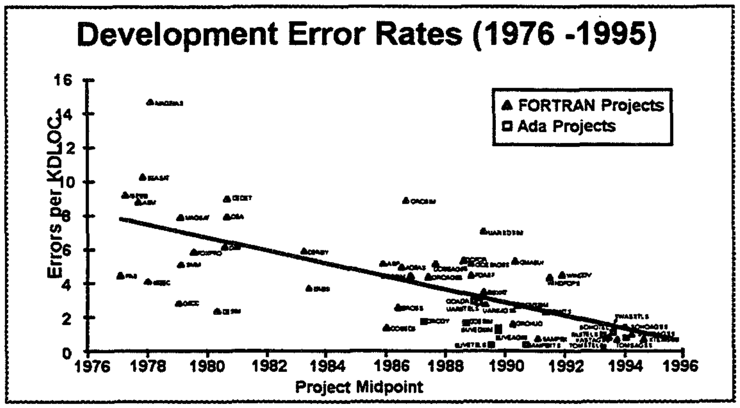
1	Improvement Goal	• Increase quality
	Baseline Measures	• 6.5 errors per KSLOC
2	Leverage Area	• Human discipline
	Process or Technology	• Various testing and review techniques • Cleanroom Methodology
3	Expectations	• Fewer errors during development and use • No additional cost
	Experiment Approach	• Iterative refinement • Controlled experiments; sequential projects
4	Deployment	• Broad use of most beneficial subset of techniques • Subset included in standard process

Quality Improvement Cycles

3 improvement cycles
Iterative refinement of existing technologies
Scope: Small to larger projects; unit to full system testing



Quality Techniques - Results



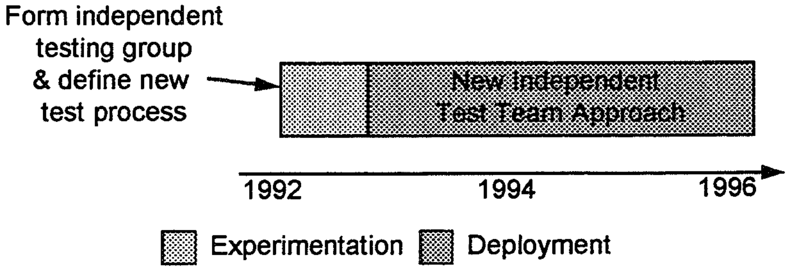
Intermediate deployment drove steady decrease in error rates.
85% improvement over 15 years.

Example 3 - Independent Testing

1	Improvement Goal	• Reduce cost and schedule
	Baseline Measures	• 254 staffmonths per mission • 106 weeks per mission
2	Leverage Area	• Eliminate process redundancy
	Process or Technology	• Form independent test teams from system and acceptance test groups • Overlap testing and development of builds
3	Expectations	• Reduced cost and schedule per mission • No loss of quality
	Experiment Approach	• Define process • Reorganize and pilot
4	Deployment	• Full use on all applications • Test teams fine-tune process

Testing Improvement Cycle

1 improvement cycle
Refinement of existing process and organization change
Scope: Piloted on all projects immediately

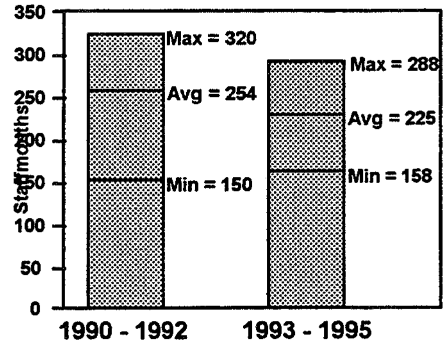


Form independent testing group & define new test process

Independent Test Teams - Results

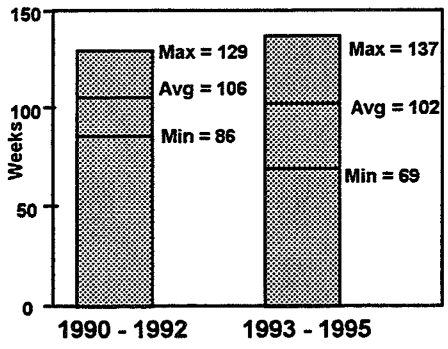
Measurements show modest improvements

Total Mission Cost



10% Cost Reduction

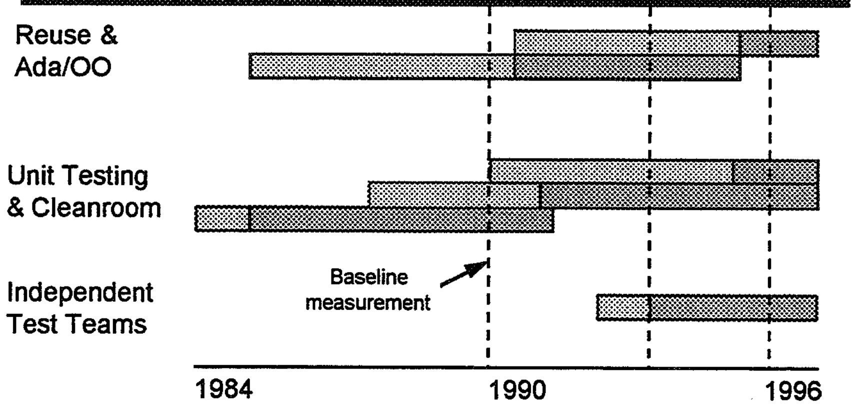
Mission Duration



5% - 20% Improvement

Simultaneous Experimentation

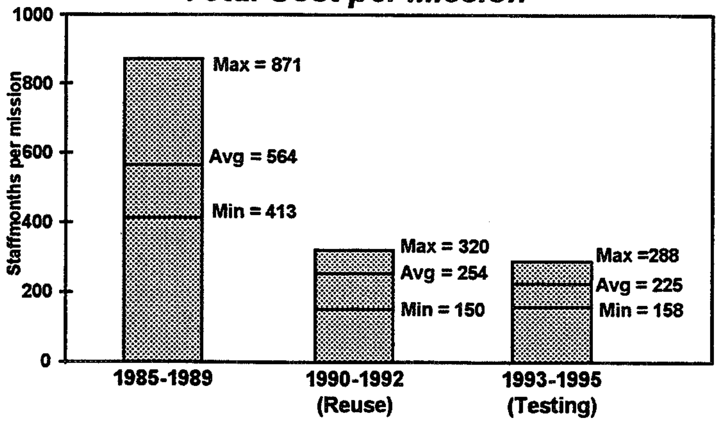
Multiple improvements were piloted simultaneously, but on separate project sets.



Overall Improvement

Improvements combine to make 60% cost reduction

Total Cost per Mission



Keys to Success

- Focus on one primary organizational goal
- Select process changes that leverage people (use technology to replace routine tasks)
- Allocate more time (iterations) when creating process from concepts
- Actively seek developer feedback

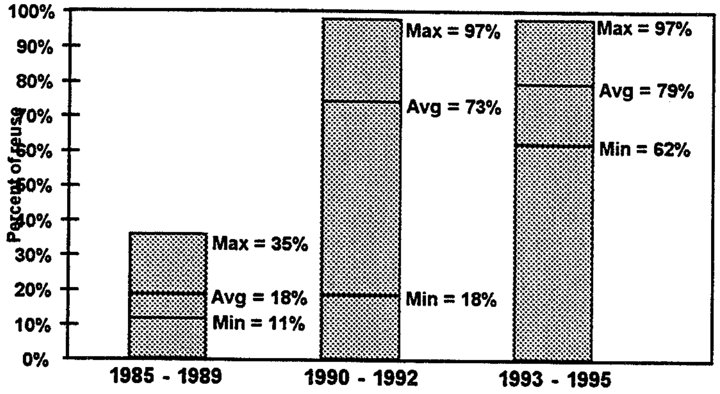
Conclusions

- More localized process changes lead to more rapid rate of improvement ....

  . . . but, broader conceptual changes result in larger improvements.

- Experimentation allows for intermediate deployment of new process or technology with minimal risk

340% Total Increase in Reuse



Evolving the Reuse Process at the Flight Dynamics Division (FDD) Goddard Space Flight Center

S. Condon,[1] C. Seaman,[2] V. Basili,[2] S. Kraft,[3] J. Kontio,[2] Y. Kim[2]

52-61
415645
360680
P 82

Abstract

This paper presents the interim results from the Software Engineering Laboratory's (SEL) Reuse Study. The team conducting this study has, over the past few months, been studying the Generalized Support Software (GSS) domain asset library and architecture, and the various processes associated with it. In particular, we have characterized the process used to configure GSS-based attitude ground support systems (AGSS) to support satellite missions at NASA's Goddard Space Flight Center. To do this, we built detailed models of the tasks involved, the people who perform these tasks, and the interdependencies and information flows among these people. These models were based on information gleaned from numerous interviews with people involved in this process at various levels. We also analyzed effort data in order to determine the cost savings in moving from actual development of AGSSs to support each mission (which was necessary before GSS was available) to configuring AGSS software from the domain asset library.

While characterizing the GSS process, we became aware of several interesting factors which affect the successful continued use of GSS. Many of these issues fall under the subject of evolving technologies, which were not available at the inception of GSS, but are now. Some of these technologies could be incorporated into the GSS process, thus making the whole asset library more usable. Other technologies are being considered as an alternative to the GSS process altogether. In this paper, we outline some of issues we will be considering in our continued study of GSS and the impact of evolving technologies.

1. Introduction

Since 1985 the Software Engineering Laboratory (SEL) has been evolving methods of software reuse through a series of studies, experiments, pilot projects, and full-fledged development projects at the Flight Dynamics Division (FDD) of NASA's Goddard Space Flight Center (GSFC). The SEL adopted Ada83 for these experiments and projects at a time when C++ was still relatively unknown. From this Ada work, the SEL determined that object-oriented (O-O) technology was providing the best reuse benefits within the FDD.

Around 1989-90 the Ada/O-O experience merged with an FDD-wide initiative to develop a "configurable" flight dynamics attitude support system. The result evolved into the *Generalized Support Software (GSS) Domain Engineering Process*. By means of this process, the FDD has shifted from developing applications to configuring applications out of generalized, reusable assets. The term "assets" encompasses design specifications, code components, tools, and standards. To date, eight applications, supporting two NASA satellite missions, have been configured from the GSS asset library and delivered to acceptance testing.

A SEL *Reuse Study* team was tasked to analyze the GSS process, determine the cost and quality of the resulting systems, document and evaluate its strengths and weaknesses, and propose modifications to it. This paper presents the preliminary results of this SEL study.

The paper examines several relevant cost issues. It compares the cost of investment in the GSS asset library to the investment in previous FDD reuse libraries. It compares the deployment costs (design, configuration and testing) of GSS-based applications to the development costs of previous FDD applications and contrasts the resulting cost savings with the investment cost in the GSS asset library. The paper also demonstrates that the GSS process has resulted in a significant decrease in the time required to field a new application.

[1] Computer Sciences Corporation, Lanham-Seabrook, Maryland

[2] Computer Sciences Dept., University of Maryland, College Park, Maryland

[3] Goddard Space Flight Center, Greenbelt, Maryland

In addition to analyzing software metrics such as effort and cycle time, the reuse study team interviewed numerous domain analysts, mission analysts, component engineers, application configurers, and application testers who have been involved in the GSS process. The study team adopted Yu's Actor-Dependency (AD) formalism to model the dependence of various GSS process *actors* on other actors and resources. In order to further understand more complex actors in this process, the team applied Yu's Agent-Role-Position (ARP) formalism to make explicit the many different roles one actor may play in the process. (Reference 1)

2. History of FDD Reuse

2.1 Environment of the FDD & SEL

Over the past decade, the FDD of GSFC has usually consisted of about 100 civil servants supported by 300-400 CSC and subcontractor personnel. (In the last two years, NASA-wide reductions in the workforce have reduced these numbers somewhat.) Of these personnel, about 40% are software developers or testers. Another 40% are operations personnel or FDD *analysts*. The analysts are the experts in orbital mechanics, mathematics, or other technical disciplines who write the software requirements for FDD applications.

The mission of the FDD is to build, deploy, and maintain space ground systems for NASA science missions, with emphasis on earth orbiting satellites. Flight dynamics applications are essentially scientific data processing systems: some are institutional (i.e., they support multiple missions) and others are mission-specific (i.e., a new one needs to be built for each new spacecraft). Each FDD application supports some aspect of spacecraft flight dynamics via one of three domains: (1) attitude determination,[4] (2) mission and maneuver planning, or (3) orbit and navigation. This paper focuses on the evolution of software reuse within the attitude determination domain of the FDD.

The SEL is a virtual organization which consists of civil servants from the software development group of the FDD, CSC contractors supporting them, and representatives from the Computer Science Department of the University of Maryland at College Park. The SEL has been in existence for over 20 years, during which time it has guided, studied, documented, and nurtured software experimentation within the FDD. (Reference 2)

2.2 History of S/W Reuse at the FDD & SEL Prior to GSS

During the last dozen years, the SEL and the FDD have focused in particular on how to increase software reuse levels, with the expectation that this would reduce cost and cycle time. At the beginning of this experimentation, the FDD was developing software applications in a FORTRAN mainframe environment, achieving a modest level of reuse of very low level utilities. Through a series of studies, experiments, pilot projects, and full-fledged development projects, the SEL and FDD began evolving methods of software reuse. Efforts were focused in the attitude determination domain, whose class of mission-specific applications would benefit most from increases in software reuse.

The SEL learned a great deal about using O-O and Ada generics for one particular type of application, a simulation test tool whose development was transferred from the IBM mainframe to an Ada-friendly platform, the DEC VAX. From these experiments and mission projects, the SEL determined that the use of object-oriented principles, rather than the Ada language itself, was providing the primary reuse benefits within the FDD. (Reference 3)

The bulk of the FDD's mission-specific applications, the AGSSs, however, continued to be developed in FORTRAN on the IBM mainframe. The SEL was unable to transfer its Ada practices to the mainframe because adequate Ada tools for the mainframe environment were lacking. In lieu of this, the FDD applied some domain engineering concepts to create two FORTRAN reuse libraries for developing AGSSs. One library was developed to support AGSSs for non-spinning satellites, and the other for spinning satellites. The majority of satellites supported by the FDD, traditionally, are non-spinning. The FDD had some success with the FORTRAN reuse libraries, but the results were not truly "generalized" and the libraries grew with each new mission and became cumbersome to maintain. Nonetheless, these were all valuable experiences on which the FDD was able to build.

[4] "Attitude" means the spatial orientation of a spacecraft

2.3 Motivation, Goals and Definition of GSS

Concurrent with the SEL-sponsored experiments in O-O, was a division-wide FDD initiative to examine the possibility of generalizing all flight dynamics software so that in future all applications would be *configured* rather than *developed.* The members of this team wrestled with what it means to "configure" an application, as opposed to "develop" an application, and came to the conclusion that it was only possible if an FDD reuse library were built around *objects.* This decision made the O-O experiments all the more important. Around 1989-90 the Ada/O-O experience and the search for "configurable" flight dynamics software applications merged and evolved into what was to become the *Generalized Support Software (GSS) Domain Engineering Process.*

The GSS process relies upon the GSS *Asset Library,* a library of generalized, configurable application components developed by the FDD with an object-oriented domain engineering approach. GSS specifications adhere to a standardized approach for specifying object-oriented classes. This standardization allows the use of standard rules for the implementation of each class, including a generic detailed design for each class and a system architecture that allows classes to be configured into a program that communicates with the FDD's User Interface and Executive (UIX). By means of the GSS process, the FDD has shifted from developing applications to configuring applications out of generalized, reusable assets. The term "assets" encompasses design specifications, code components, tools, and standards.

In 1992 the design of the GSS asset library got into full swing, followed in early 1993 by coding of the assets, which were implemented in the Ada83 language and resided on a DEC Alpha workstation. In February 1995 work began in earnest on configuring the first application from this asset library. To date, eight applications, supporting two NASA satellite missions, have been configured from the GSS asset library and delivered to acceptance testing. These applications run on HP or Sun workstations.

2.4 GSS as an Experience Factory

In order to carry out process improvements within the FDD, the SEL functions as an *experience factory* in relation to the *project organization.* The project organization consists of FDD mission analysts, application developers, and application testers. The mission analysts are the FDD personnel whose training and experience in orbital mechanics and mathematics qualifies them to write the requirements for FDD applications. As the project organization goes about its business of developing applications, the experience factory collects metrics and lessons learned from them. The experience factory staff stores these data in a database, analyzes the data, suggests and conducts additional experiments, and finally packages these distilled project organization experiences into recommended best practices, estimation models, and software development training courses, which spread these process improvements throughout the FDD project organization. Figure 1 depicts this traditional relationship between the project organization and the experience factory. A heavy dashed line separates the two groups. The light dotted line separating the mission analysts from the software developers on the project organization side reflects the fact that traditionally the SEL has not collected metrics from mission analysts in the FDD.

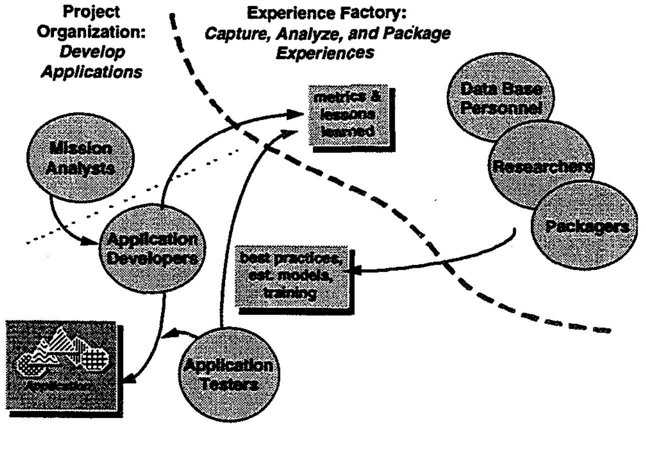

Project Organization: Develop Applications
Experience Factory: Capture, Analyze, and Package Experiences
metrics & lessons learned
Data Base Personnel
Mission Analysts
Researchers
Packagers
Application Developers
best practices, est. models, training
Application
Application Testers

Figure 1: Traditional SEL Experience Factory

With the development of the GSS Asset Library, the boundaries and scope of the experience factory appear to have expanded. New personnel, formerly part of the project organization, are now fulfilling experience-factory-type roles. Instead of supplying only *process improvements* to the FDD project organization, however, these people are also supplying *product improvements* to the FDD in the form of generalized library assets.

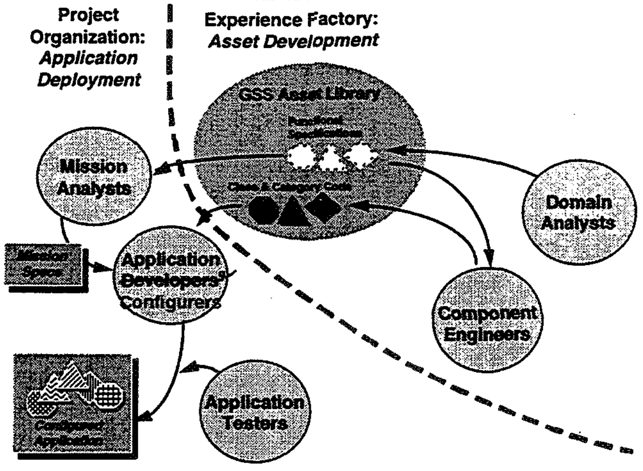


Figure 2. GSS Component Development and Application Deployment Process

Figure 2 depicts this new dimension to the experience factory concept at the FDD. A few former mission analysts have become *domain analysts*. They have designed the GSS architecture and written the GSS functional specifications for the library assets. At the same time several applications developers have become *component engineers* and have coded the classes and categories defined by the GSS functional specs. With these assets developed, the project organization then follows a streamlined process for application deployment. Under the new deployment process, a mission analyst must write the GSS *mission specification* that stipulates which GSS classes & categories are required for the application, which of the many parameters associated with these assets are necessary for this application, and what values need to be assigned to these parameters. This mission specification is passed to an *application configurer*—application developers are no longer needed—and the configurer then instantiates the specified objects from the generalized classes in the asset library and links them to form the desired application. The application testers then test the application and turn it over to operations.

3. Characterization of the GSS Application Deployment Process

A SEL *Reuse Study* team was tasked to analyze the GSS configuration process, determine the cost and quality of the resulting application systems, document and evaluate the strengths and weaknesses of the process, and propose improvements to it. In this section, we describe the preliminary results of this study of the GSS configuration, or application deployment, process, which is used to define, configure, and test an attitude support software application. Below, we describe the methods we used to gather and analyze this process information. In the sections which follow, we first characterize the configuration process quantitatively with respect to its cost, schedule, and the errors in the resulting applications. We then present the process graphically and analyze its inner workings.

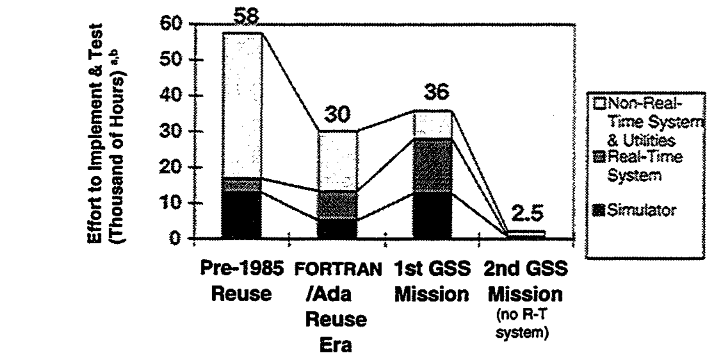
To model the GSS configuration process, the team began by studying documentation and holding informal discussions with managers, task leaders, and a few key technical personnel. At the same time we began to analyze SEL data on effort, estimates, schedules, and software changes related to the GSS asset library and to the software applications that were configured from it. As this metrics data analysis was proceeding, we conducted numerous detailed, structured interviews with people playing a variety of roles related to GSS in order to obtain information of sufficient detail to model the configuration process.

3.1 Analysis of Metrics Data

3.1.1 GSS Costs

There are two relevant costs to consider when evaluating the GSS project. One is the cost associated with configuring applications from GSS components. Figure 3 compares the cost of deploying GSS-based applications to costs in the previous two eras, and demonstrates that GSS-based applications can be deployed for as low as 10% of the cost required during the FORTRAN/Ada reuse era.

Prior to 1985 it cost 58,000 hours to develop and test the attitude support applications for a typical FDD mission. Later, when the FDD was using Ada reuse libraries to develop simulators and FORTRAN reuse libraries to develop AGSSs, this cost dropped to 30,000 hours per mission. In both eras the development of the non-real-time system and the utilities required the most effort.



[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.
[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

Figure 3: Reduced Deployment Costs Due to GSS Process

When it came time to support the first mission with the GSS library, the simulator was configured first, and the real-time portion of the AGSS was configured second. In each case, the GSS asset library was still undergoing redesign and growth. The configurers were also evolving the configuration process. Consequently, the cost of deploying these first two applications was more than it had been in the FORTRAN/Ada reuse era. When the time came to configure the non-real-time portion of the AGSS and the utilities, the asset library and configuration process had stabilized. As a result, this cost only a fraction of the typical cost from the previous era. With the second GSS-supported mission, we see even more dramatic savings. The simulator and the non-real-time system plus utilities each cost on the order of 10% of their cost from the FORTRAN/Ada reuse era. No real-time system was required for this application.

The other important cost to remember is the initial cost of building the GSS library itself. These costs are shown in Figure 4 alongside the costs to develop and test the FORTRAN and Ada reuse libraries from the previous era. For the GSS asset library we know that the domain analysts spent 36,000 hours defining the requirements and the *logical* design in the GSS functional specifications. The component engineers spent 40,000 hours creating the *physical* design and implementing, inspecting, and unit testing the generalized Ada83 classes and categories. We know the effort required to develop and test the FORTRAN and Ada reuse libraries, but we do not know the hours spent on requirements, since traditionally the SEL does not collect metrics from FDD mission analysts. Even so, we can see that the GSS library was developed for less than the combined cost of developing the FORTRAN and Ada reuse libraries, which it replaced.

Figures 3 and 4 further demonstrate that if the FDD continues to deploy GSS-based applications for 10% of the cost of the preceding era, the FDD will recoup its entire library investment cost of 76,000 hours by

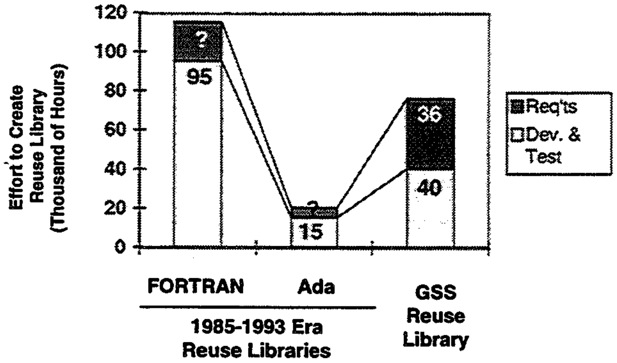
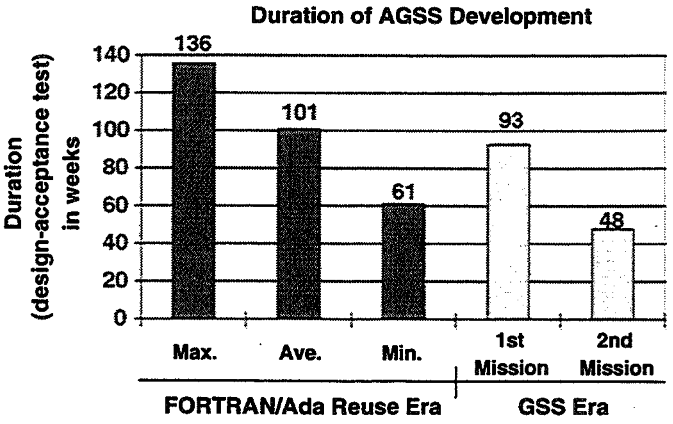


Figure 4: Library Investment Costs in Two Eras



Note: GSS era estimates assume project completions by 1/30/97

Figure 5: GSS Reduces Deployment Cycle Time

the fourth GSS supported mission.

3.1.2 Application Deployment Cycle Time

The GSS process has resulted not only in a great reduction in the cost of deploying an application, but also in a significant reduction in the cycle time required to deploy an application. Figure 5 reveals that the time to field an AGSS during the FORTRAN reuse era ranged from 61 to 136 weeks, with an average of 101 weeks. The time required to design, configure, and test the applications for the first GSS-supported mission was a little less than the average for the preceding era. The second project, however, was completed in less than half of the average cycle time for the FORTRAN/Ada era. In fact, it took less time than any project in the previous era. It seems likely that project duration can be further reduced with this reuse process.

3.2 Process Diagrams

After gaining an initial understanding of the GSS environment and how it is used, the team developed a detailed interview guide and conducted structured interviews with most of the designers, developers, configurers, and testers involved in the GSS processes. Once a sufficient body of information had been collected, we began to organize it by modeling the relevant processes, in particular the GSS configuration process.

We chose to use Yu's Actor-Dependency (AD) model to portray the interactions, roles, and dependencies between the actors in the GSS processes. Figure 6 is an AD model reflecting the same level of detail as depicted in Figure 2. The AD diagram reflects how each team depends on other teams. The types of dependencies are

- resource dependencies (depicted by a rectangle), which indicate that the depender relies on some artifact, document, or information from the dependee;
- task dependencies (depicted by a hexagon), which indicate that the depender relies on the dependee to complete some defined set of steps. The dependee may or may not be aware of the goals of this task;
- goal dependencies (depicted by an oval), which indicate that the depender relies on the dependee to achieve some well-defined goal. The depender has a great deal of freedom to determine how to reach that goal; and
- soft goal dependencies (depicted by a distorted oval, i.e., a "peanut" shape), which indicate that the depender relies on the dependee to achieve some goal which is not well-defined, i.e. the depender and dependee may not agree on, and must negotiate, exactly how the goal is to be satisfied.

The following AD diagrams focus more on the GSS application configuration process and show the relevant roles and dependencies at a lower level of detail.

Figure 7 expands the complex social actors of Figure 6 into their substructure of agents, roles, and positions. Agents are actual, physical people and groups of people that actors represent. Roles indicate what parts of the process an actor is involved in. Positions are the organizational titles and jobs that an actor holds. Positions generally “cover” one or more roles, while roles are “played” by an agent, who also “fills” one or more positions. In Figure 7, only some of the relevant dependencies are shown and (for the most part) are not identified by type in order to simplify the diagram.

Figure 8 shows, at a high level, the sequences of tasks that must be completed in order to configure a GSS application, and the inputs and outputs of those tasks. Tasks are represented as ovals and artifacts (inputs and outputs) as rectangles. Many of the tasks refer to task dependencies in Figure 6.

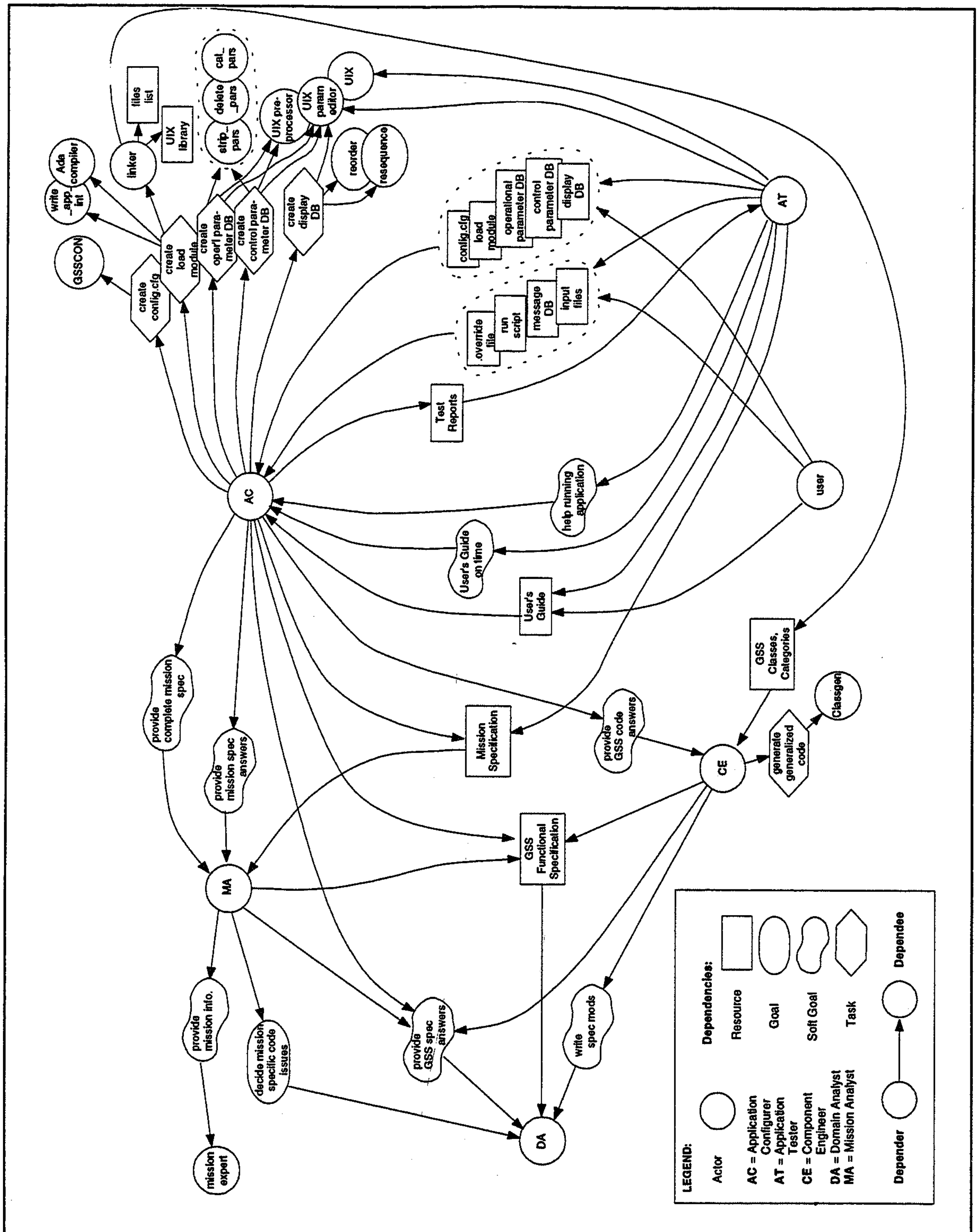


Figure 6. Actor-Dependency (AD) Model of GSS Application Deployment Process

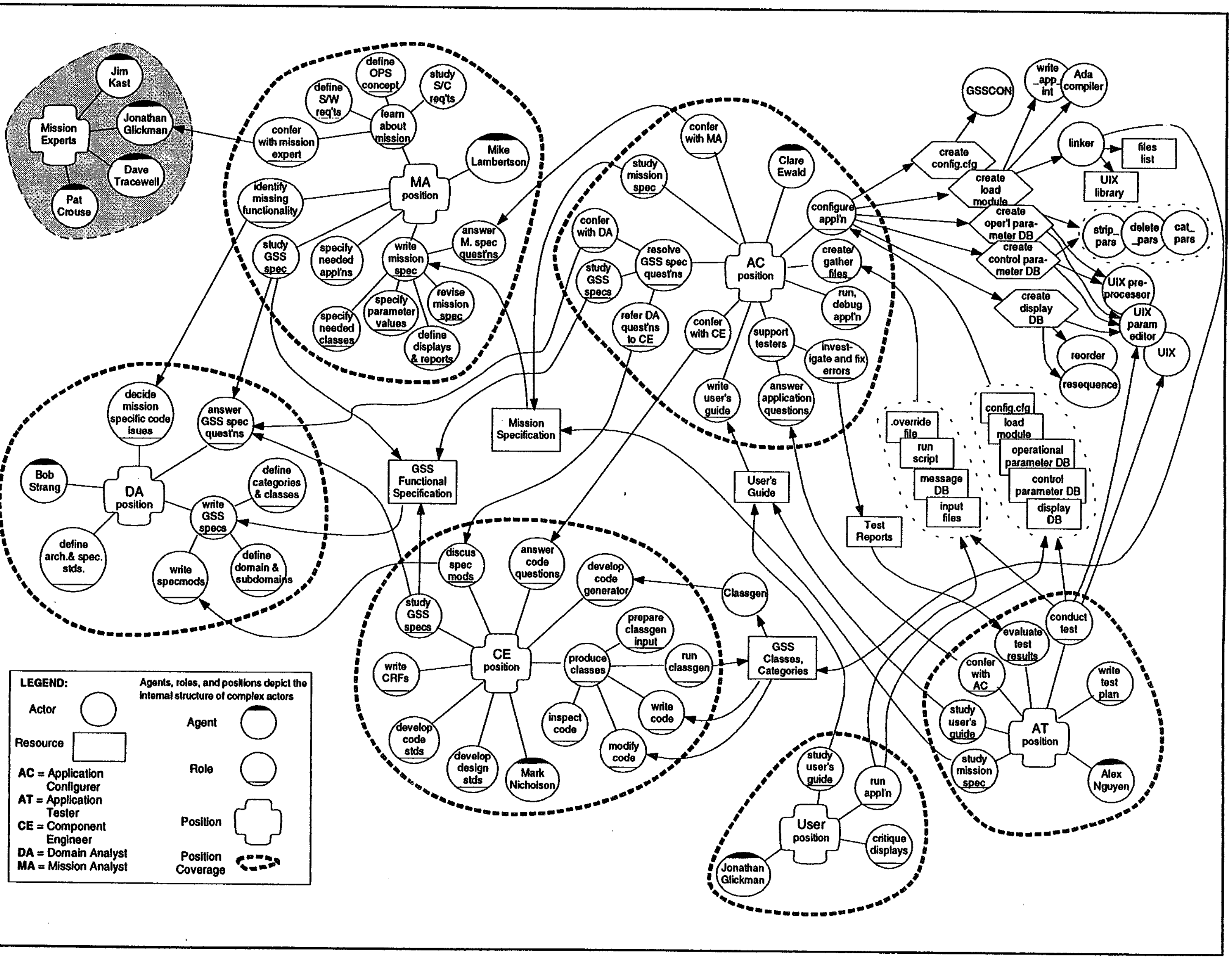


Figure 7: Agent-Role-Position (ARP) Model for GSS Application Deployment Process

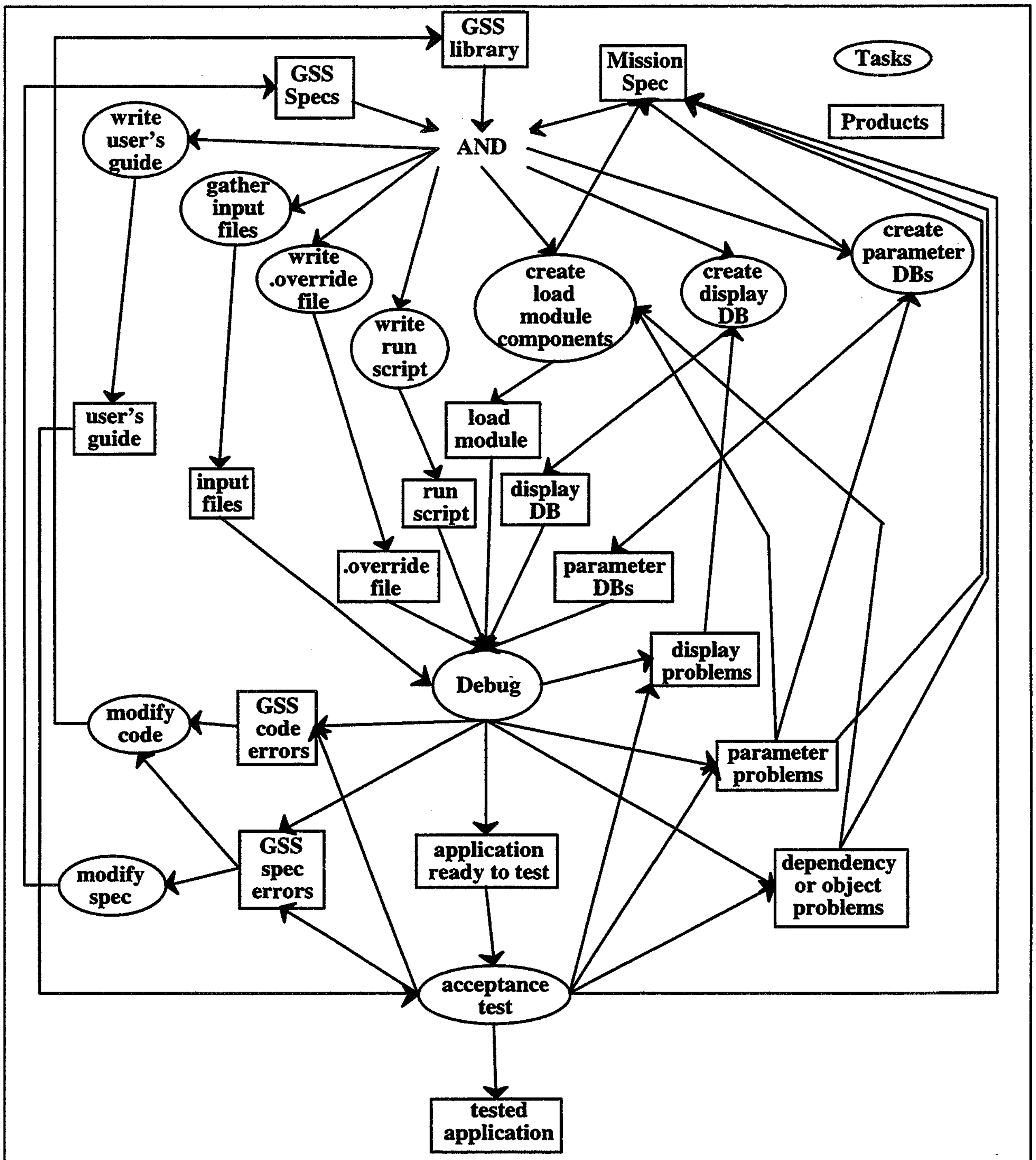


Figure 8: GSS Configuration Tasks

4. Recommendations for Improvements to the GSS Configuration Process

As is often the case, organizational and technical details which were overlooked at the project's inception have come back in various forms to threaten the full success of GSS. Despite dramatic reductions in application deployment cost and cycle time, the GSS process has not won the full support of all groups within the FDD. Although FDD management mandated that software developers and analysts would jointly design the GSS process, the resulting process is today viewed by many as the child of the software developers, with less than full partnership from the analysts.

But this is more than merely a perception. The current GSS process provides a good tool that allows traditional software developers to quickly configure flight dynamics software applications. At the same time, however, the current GSS process contains hurdles for mission analysts, whom FDD management would like to see making more direct use of the GSS. This is because the GSS process and the GSS documentation are inherently more understandable to the GSS developers and configurers than to the majority of FDD mission analysts. As discussed later, the writing of the initial mission specification in particular is a task logically performed by mission analysts, but at this time it requires a very technical level of understanding of GSS. This level of understanding is very difficult, and not necessarily appropriate, for analysts to achieve. As a result of this, relatively few FDD analysts are currently involved in the GSS process.

As a result of our in-depth characterization of the GSS configuration process, we discovered several opportunities for improvement. Some of these were synthesized from the comments of several interviewees, while others came directly from GSS developers, configurers, and testers. Most relate to the problem described above (of the barriers to use by analysts), but also would improve the GSS process in other ways as well.

4.1 Storing application requirements

Several problems were cited that might be ameliorated by storing the information contained in the mission specification in database form. First of all, it would facilitate the reuse of requirements, which is common from one application to another. Instead of manually editing reused parts lists, display files, parameter files, etc., database operations could be used to modify these elements in the database to help ensure consistency and avoid errors.

Secondly, it has been stated as a goal of GSS that eventually mission analysts should be able to configure attitude software with little or no intervention from GSS developers. There are several barriers to achieving this goal, one of which is that the writing of the mission specification seems to require very specialized skills. This is more than a user interface problem, but using a database format rather than a textual one may help.

Designing and maintaining a database for mission and application requirements would not be a simple task. It would require the borrowing or hiring of a specialist in database design, and a careful analysis of the needs that the database is meant to satisfy. Because of some of the points discussed above, a database system with an adequate user interface is especially important. Also, it would be helpful to be able to integrate this database with other databases used in the environment, e.g. databases used to store new component information.

4.2 Automatic generation of configuration inputs

Another advantage of storing mission-specific information in a database is that it would facilitate the automatic generation of some of the inputs to the GSS configuration. Generating these files at present is tedious and time-consuming. Writing the parts list in particular has been described as a translation of the mission specification from one notation to another. Such a translation could be automated if the mission specification were stored electronically. Even better, the tools which process the parts list could be rewritten so that they access the database directly. As mentioned later, such a database could also facilitate the automatic generation of some parts of the user's guide. Also, it is conceivable that a database of application requirements could also be used to automatically generate the artifacts needed as input to UIX (the user interface facility), including the display files, the parameter files, and the message files.

4.3 Support for learning GSS

As mentioned earlier, the specialized skills required for writing mission specifications seem to be a barrier to making GSS usable by mission analysts. Making the mission spec database-based rather than a textual document may help somewhat. However, it does not solve the root problem, which is that writing the mission specification involves choosing the proper configuration of GSS components for a particular mission. This requires a level of understanding of the GSS architecture that, up until now, mission analysts have been unable or unwilling to attain. This problem has both organizational and technical aspects. Analysts were not involved enough in the development of GSS to give them any sense of ownership. Thus, they are not highly motivated to take the time necessary to learn to use GSS. Motivation is further inhibited because, up until now, one particular analyst has been willing to take on the task of writing mission specifications for all missions using GSS-based software. From a technical point of view, the current documentation on GSS (the GSS functional specifications) are written by and for software developers, not mission analysts. Their size and technicality are daunting, to say the least, and their organization is closely tied to the organization of the software, which is not necessarily the most logical from a user's point of view.

Thus, if GSS is to achieve the goal of being fully usable by mission analysts, a serious effort must be made to support learning. There is a growing area of research and development in software engineering in object-oriented frameworks; for example, the SEL is studying learning and reading techniques for frameworks (Reference 4). GSS fits the definition of an O-O framework, which is a domain-specific repository of software classes which fit into a cohesive architecture designed specifically for the domain. To the best of our knowledge, GSS is the only O-O framework specific to the flight dynamics domain. However, much of what has been learned about how to support the learning of frameworks in other domains could be applicable here. A number of strategies have been used: cookbooks of application templates and variations, example applications, documented class hierarchies, etc. One approach may be to develop a scenario-driven overlay for the GSS functional specifications which helps organize the specifications according to user scenarios. Many of these techniques could be useful in helping mission analysts understand GSS sufficiently to begin producing their own applications.

Designing learning support materials for GSS would involve some experimentation to determine which strategies are most helpful for mission analysts. This would require some investment of time and resources, and a serious commitment to finding an appropriate solution for the FDD domain and organization. It is also crucial that the support materials are designed for the most part by mission analysts, not software developers. The involvement of members of the analyst branch of FDD is necessary to ensure that the materials, and GSS, will be used in the future.

4.4 User's Guide

User's guides are required to be delivered to the acceptance testers with the application, but they are usually not completed until well after that point. Testers usually do not have them available in time to help with testing at all. Instead, they rely for the most part on the mission specification. However, the testers did not seem to see this as a big problem. The configurers, on the other hand, were not highly motivated to write user's guides and it was treated as a necessary but low-priority chore. A suggested improvement, then, is first to determine what information is really useful in the user's guide (for both testers and eventual users), then to investigate the possibility of automatically generating parts of the user's guide from the mission specification (this might be facilitated by the database suggested earlier), and finally, if necessary, assign a qualified technical writer to take on the writing of user's guides, as a task apart from configuration of the application.

5. New Directions for Reuse Study

Having characterized the GSS process, the Reuse Study Team will concentrate in the coming months on putting this process into perspective, particularly with respect to its changing technical and organizational context. First of all, a number of technological advances have taken place in software engineering since the inception of GSS. These advances may be relevant to how GSS is used in the future. Furthermore, some developments in the marketplace have produced alternative approaches to reuse. Some of these may be appropriately used instead of GSS in some cases. The focus of the Reuse Study Team in the near term will be to study which of these emerging technologies could best be incorporated into GSS and how, and under what conditions GSS could be supplanted with technology that is now available elsewhere. We hope to evolve guidelines to be used by FDD mission teams in choosing how best to produce their software applications. In the sections below we outline some of the issues on which we will concentrate.

5.1 Evolving Technologies

Over the years that the GSS has been evolving, many technologies have been evolving in the marketplace. Some of these technologies require a second look to see how they compare to the GSS process today. It may be that the GSS process could benefit from incorporating some of these technologies.

5.1.1 Object Orientation

The GSS assets have been built from an object-oriented perspective since its inception. In many ways, the development of GSS was ahead of its time, in that tools and techniques for developing object-oriented systems were not available when the GSS team needed them. For example, the only object-oriented programming languages that were available at the inception of GSS were Ada83 and Smalltalk. Now, other languages are available, such as C++ and Ada95, along with supporting tools. We will consider whether or not GSS suffered from not having these languages and tools available, and if any of the currently available languages and tools might be useful in the future maintenance of GSS. The software engineering field also knows more now about such topics as object-oriented design, testing, and maintenance. New advances need to be examined to determine their applicability to GSS.

5.1.2 Graphical User Interfaces

A User Interface and Executive (UIX) was developed by a separate group of FDD developers, in parallel with GSS, to provide GUI capability for GSS-based applications. It was decided to develop the GUI capability in-house because, at that time, no appropriate GUI packages were available in the commercial market. That is no longer the case, so it is appropriate to compare UIX to what is currently available commercially, off-the-shelf (COTS). It may be cost-effective to replace UIX with a more user-friendly and robust GUI capability developed elsewhere.

5.1.3 Other COTS Products

To support the GSS process, a number of tools have been developed in-house, such as code generators and editors. Most of these were developed in an *ad hoc* (as needed, as time permitted) manner. As the sophistication and quality of currently available COTS products has risen, we will investigate whether some could be used to support the GSS process. Some COTS products may even be appropriate to replace the GSS process in some cases, as discussed below.

5.2 Alternative Reuse Processes

For several years, the FDD has been slowly developing more and more software on UNIX workstations and weaning itself from its traditional reliance on the IBM mainframe. In the 1990s the FDD began to develop some of its attitude support software for execution on UNIX workstations rather than on the IBM mainframe computer. For example, the AGSSs supporting the three most recent operational satellites (SOHO, SWAS, and XTE) ran partly on the IBM mainframe and partly on the UNIX workstations. Since the FORTRAN reuse libraries resided only on the mainframe, the subsystems based on the workstations had to be written essentially from scratch. The GSS strategic reuse library was designed entirely for UNIX workstations, and would have been useful for these subsystems, but it was not yet available.

The movement from the mainframe to workstations received a big impetus near the end of fiscal year 1995, when FDD management mandated that all software would be removed from the IBM mainframe computers by the end of fiscal year 1996. Consequently, much of the institutional and mission-specific FORTRAN code on the IBM mainframes needed to be ported to workstations in a hurry.

It was initially decided that the mainframe portions of the three most recent operational AGSSs would be re-implemented on the workstations by configuring them from the GSS library. In order to continue supporting the older legacy missions, however, an alternative method was sought. Since these AGSSs were built primarily from the FORTRAN reuse libraries and ran entirely on the mainframe, it was decided to port these libraries to the workstations.

The FORTRAN reuse library used for supporting non-spinning satellites was rehosted by two mission analysts with considerable support from some COTS products. FORTRAN subroutines were edited using word processors in order to conform to language restrictions of the COTS products. The analysts followed some process shortcuts and made liberal use of certain language features provided by the COTS products. During this rehost, the library specifications were not rigorously followed and were not updated to reflect the rehosted version of the library. Another FORTRAN reuse library, used to support spinning satellites, was rehosted by software developers, using the same COTS products. However, they closely followed the library specifications and made little attempt to take advantage of language features unique to the COTS products.

The analysts who rehosted the first library enjoyed using the COTS product and demonstrated that the rehost could be done cheaply and quickly. They found that they had a lot of control over the process and were able, because of their position, and/or the features of the COTS products, to rapidly make changes to the library during the rehost. As a result of their favorable experience, the rehosted libraries, together with their COTS umbrella, are now viewed as an alternative process for supporting new FDD missions as well as legacy missions.

In addition to these COTS products used for rehosting attitude determination systems, there are additional COTS products that can meet various other parts of typical FDD mission requirements. Some of these products are already being reviewed and adopted to support mission/maneuver planning and orbit/navigation requirements for upcoming FDD missions.

The Reuse Study Team has been charged with studying the processes associated with the maintenance and reuse of GSS, as well as those that utilize the rehosted FORTRAN reuse libraries in the development of mission support software. Our work thus far has resulted in a detailed understanding of the GSS configuration process, described in the previous sections. As well, we have come to some understanding of the questions around which to focus this comparison. These questions represent some points of disagreement between COTS and GSS proponents, some concerns raised by developers and users of both approaches, and our own analysis of interview data. These questions are presented in the sections below.

5.2.1 User Interface

GSS uses a unified user interface called UIX for all applications. UIX was developed in-house, in parallel with GSS. This has caused some problems in the testing of GSS, when errors turn out to be UIX errors, not errors in the GSS code. The use of UIX also requires the handling and formatting of a number of large files (parameters, displays, messages) in configuring an application, which can be tedious and error-prone.

Many COTS products provide their own GUI capability, which is used to create a user interface for each application. This interface is not necessarily consistent.

How important is a unified user interface? How difficult would it be to unify all the COTS-based user interfaces?

5.2.2 Is Object-Oriented Technology Superior?

The rehosted libraries are written in a procedural language associated with the COTS products used to support the rehost, in some cases from scratch and in others converted from FORTRAN code using a text editor. GSS applications are mostly Ada83 with a small amount of C code in some cases. Thus, the GSS library is based on O-O concepts, whereas the rehosted libraries, and their related applications, are not. Prior to GSS, the SEL determined that the use of Ada and O-O concepts in the FDD resulted in smaller systems to perform more functionality, while the FORTRAN reuse libraries continued to grow in size.

Since they are based on FORTRAN, will the rehosted reuse libraries continue to have the same disadvantages (in particular, code growth) as did the original FORTRAN libraries? If so, this makes the FORTRAN libraries a less attractive choice compared to O-O Ada reuse libraries. Or is there some attribute of the COTS products or the rehosting process which mitigates these disadvantages?

5.2.3 Software Engineering Practices

The design of the rehosted libraries relies heavily on the use of Global COMMON data. The software elements of the resulting applications are very tightly coupled to these data structures. Also, as mentioned earlier, one of the rehosted libraries has a code structure which mirrors the original FORTRAN structure very closely. Some developers also expressed concern that the rehosting efforts did not follow standard software engineering practices, such as inspections. On the other hand, it could be argued that rehosting does not warrant such a high process overhead because it is based on software that has been in operation for a long time.

GSS, on the other hand, was developed in accordance with more modern O-O concepts and practices. A rigorous software engineering process was followed, including design and code inspections and rigorous testing.

Does the use of O-O concepts and software engineering practices really make a difference in this case? Or does the fact that the rehosted software is based on such a time-tested library make up for its deficiencies in this area?

5.2.4 Maintenance

Both FDD COTS users and GSS proponents stress the advantages of their respective approaches for maintenance. The systems based on the rehosted libraries are argued to be easily and quickly modified by someone who is familiar with the domain, but not necessarily with software development. That is, an analyst does not have to rely on a software developer to make every change required. Using a GSS-based application, on the other hand, requires a delay whenever a change is requested, often until the next release of the GSS library. Thus using the MATLAB-based rehosted libraries provides users much quicker turnaround time on modifications of the application than does using GSS.

GSS proponents argue, on the other hand, that any system will degrade over time if it is allowed to be changed unsystematically by users. Also, the structure of GSS was designed to facilitate change without adding complexity or large amounts of new code.

Is it more important for the user to have quick turnaround on requested changes, or to manage the evolving structure of the software? Is there a reasonable compromise between the two? Do the COTS-based applications become more difficult to maintain the larger the application is? Does the design of GSS really ensure that it will not degrade over time? Are developers and analysts using different time scales (i.e., "quick" is 1 hr. for an analyst, but 1 day for a developer?)? Are developers and analysts looking at different scopes of the modification process (i.e., a developer looks at how quick it is to change the code, whereas an analyst looks at how long he has to wait to get the revised)?

5.2.5 Performance

The applications based on the rehosted libraries are interpreted, not compiled. In some cases the source code was automatically converted to C, then compiled. This compilation step improves processing speed by a factor of two, but still remains slower than traditional FDD applications. How much slower are the COTS-based applications than GSS-based applications, and is this difference noticeable or important to users?

5.2.6 Reliability

The AGSSs based on the rehosted libraries rely heavily on the intrinsic capabilities of the underlying COTS software for performing a number of mathematical manipulations. Care must be given to separate out errors in the COTS software from errors in the custom developed portions of the code. GSS components, on the other hand, have exhibited very low defect levels in acceptance testing. No applications of either approach, however, have been operational for long enough to assess field reliability.

What assurances do we have of the reliability of COTS products? How can it be assessed?

5.2.7 Portability

The applications based on the rehosted libraries are all designed to be part of a single system using the GUI provided by the COTS product used in the rehost. This makes porting the components relatively easy for any target platform which supports that product. On the other hand, there were some difficulties recently in porting one of the GSS-based AGSSs from the HP to the Sun workstations because UIX (the user interface which GSS uses) had not previously been ported to the Sun.

How important a criteria is portability? Can UIX and GSS be made more portable in the future?

5.2.8 Documentation

During the porting of one of the FORTRAN libraries, the original FORTRAN code structure was followed very closely. Thus, the original specifications for the FORTRAN software are still valid for the rehosted version. However, none of the advanced features of the COTS products were used which would have allowed a more efficient restructuring of the code. These features were used heavily in the porting of the other FORTRAN reuse library. As a consequence, the code is more compact than it was, but the original software specifications are no longer valid and no new specifications have been written. The analysts who were responsible for porting the libraries believe that, to a certain extent, a separate specifications document becomes less necessary because in the programming language used (associated with the underlying COTS products), the equations are written exactly as they would be written in the specification.

The design of the GSS system is documented in the GSS functional specifications, but these are 1600 pages long and, as mentioned earlier, are a real barrier to understanding the system for its eventual intended users, mission analysts. However, they seem to provide all relevant information necessary for maintaining the GSS components, and are written from a software developer's point of view.

Is either type of documentation sufficient for operation and maintenance purposes? Is the COTS-based code really self-documenting enough for maintainers to correctly make modifications? Can users of GSS components and applications be taught to use the GSS specifications effectively?

6. Conclusions

This paper presents the interim results from the SEL's Reuse Study. The team conducting this study has, over the past few months, been studying the GSS domain asset library and architecture, and the various processes associated with it. In particular, we have characterized the process used to configure GSS-based attitude ground support systems to support FDD missions. To do this, we built detailed models of the tasks involved, the people who perform these tasks, and the interdependencies and information flows between these people. These models were based on information gleaned from numerous interviews with people involved in this process at various levels. We also analyzed effort data in order to determine the cost savings in moving from actual development of AGSSs to support each mission (which was necessary before GSS was available) to configuring AGSS software from the domain library.

While characterizing the GSS process, we also became aware of several interesting factors which affect the successful continued use of GSS. Many of these issues fall under the subject of the evolving technologies, which were not available at the inception of GSS, but are now. Some of these technologies could be incorporated into the GSS process, thus making the whole asset library more usable. Other technologies are being considered as an alternative to the GSS process altogether. In this paper, we outline some of issues we will be considering in our continued study of GSS and the impact of evolving technologies.

7. References

1. Yu, E., "An Organizational Modeling Framework for Multi-Perspective Information System Design," [Conference currently unknown], 1993(?).

2. McGarry, F., R. Pajerski, G. Page, S. Waligora, V. Basili, M. Zelkowitz, *An Overview of the Software Engineering Laboratory*, Software Engineering Laboratory, SEL-94-005, December 1994

3. Waligora, S., J. Bailey, M. Stark, *Impact of Ada and Object-Oriented Design in the Flight Dynamics Division at Goddard Space Flight Center,* Software Engineering Laboratory, SEL-95-001, March 1995

4. Basili, V., G. Caleiera, F. Lanubile, F. Shull, "Studies on Reading Techniques," *Proceedings of the Twenty-First Annual Software Engineering Workshop,* Greenbelt, MD, December 1996

8. Other Sources

Boland, D., L. Cisney, S. Godfrey, S. Green, T. Gwynn, J. Langston, *Upper Atmosphere Research Satellite (UARS) Attitude Ground Support System (AGSS) Software Development History,* Flight Dynamics Division/GSFC, FDD/552-90/092, November 1990

Briand, L., W. L. Melo, C. Seaman, V. Basili, "Characterizing and Assessing a Large-Scale

Software Maintenance Organization," ICSE'95, Seattle, WA, 1995.

Brown, C., R. Coon, J. Langston, D. Spiegel, T. Wood, *Internatinal Solar Terrestrial Physics (ISTP) Program/Global Geospace Science (GGS) Project, WIND and POLAR Spacecraft Flight Dynamics Support System (FDSS) Software Development History,* Flight Dynamics Division/GSFC, 552-FDD-93/008R0UD0, March 1993

Condon, S., M. Regardie, M. Stark, S: Waligora, *Cost and Schedule Estimation Study Report,* Software Engineering Laboratory, SEL-93-002, November 1993

Coon, R., J. Golder, S. Green, J. O'Neill, *Internatinal Solar Terrestrial Physics (ISTP)/Collaborative SolarTerrestrial Research (COSTR) Initiative, Solar and Heliospheric Observatory (SOHO) Mission Attitude Ground Support System (AGSS) Software Development History,* Flight Dynamics Division/GSFC, 552-FDD-95/026R0UD0, November 1995

FDD analysts, developers, and testers, interviews with

FDD/GSFC, *MTASS FDSS Overview, Revision 1, Update 1,* October 1995

Green, D., T. Gwynn, G. Moschoglou, M. Regardie, L. Lindrose, A. Calder, S. Valett, *X-Ray Timing Explorer (XTE) Submillimeter Wave Astronomy Satellite (SWAS) Utilities Software Development History,* Flight Dynamics Division/GSFC, 552-FDD-96/007R0UD0, October 1996

Gwynn, T., M. Mills, M. Regardie, T. Rogers, *Submillimeter Wave Astronomy Satellite (SWAS)/X-Ray Timing Explorer (XTE) Attitude Ground Support System (AGSS) Software Development History,* Flight Dynamics Division/GSFC, 552-FDD-95/024R0UD0, September 1995

Kulp, D., P. Myers, M. Regardie, *Total Ozone Mapping Spectrometer-Earth Probe (TOMS-EP) Attitude Ground Support System (AGSS) Software Development History,* Flight Dynamics Division/GSFC, 552-FDD-94/031R0UD0, September 1994

MathWorks Web Site, http://www.mathworks.com/ and http://www.mathworks.com/matlab.html

NASA/GSFC Software Engineering Laboratory (SEL), *The Generalized Support Software (GSS): A Description of Its Current Software Development Process,* February 1996

Software Engineering Laboratory: data from its database

Spiegel, D., J. Doland, *Fast Auroral Snapshot Explorer (FAST) Attitude Ground Support System (AGSS) Software Developement History,* Flight Dynamics Division/GSFC, 552-FDD-94/040R0UD0, September 1994

21st Annual Software Engineering Workshop
December 4, 1996; Greenbelt, Maryland

Evolving the Reuse Process at the Flight Dynamics Division (FDD) Goddard Space Flight Center

Condon, Seaman, Basili, Kraft, Kontio, & Kim

Authors, Addresses & Affiliations

- Steven Condon[1] -- scondon@csc.com
- Carolyn Seaman[2] -- cseaman@cs.umd.edu
- Vic Basili[2] -- basili@cs.umd.edu
- Stephen Kraft[3] -- steve.kraft@gsfc.nasa.gov
- Jyrki Kontio[2] -- jkontio@cs.umd.edu
- Yong-Mi Kim[2] -- kimy@cs.umd.edu

[1] Computer Sciences Corporation

[2] Computer Science Dept., University of Maryland, College Park

[3] Goddard Space Flight Center

Outline

- Reuse History at FDD & SEL
- Evolving Reuse: *The GSS Process*
- Advantages and Issues of GSS Process
- Potential Improvements for the GSS Process

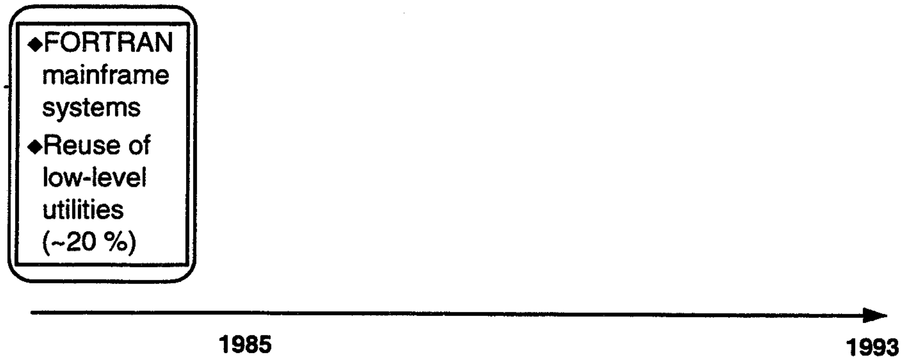
- Evolving Technologies
- Alternative Reuse Process
- Understanding Alternative Reuse Processes

- Conclusions

FDD Environment

- Size: 100 civil servants, 300-400 contractors
- Mission: Deploy mission-critical applications for NASA space ground systems
- 3 Software Domains
  - Attitude Determination
    - 200-300 KSLOC attitude ground support systems (AGSS)
    - 40-70 KSLOC telemetry simulators
  - Mission/Maneuver Planning
  - Orbit and Navigation

Reuse History at FDD & SEL

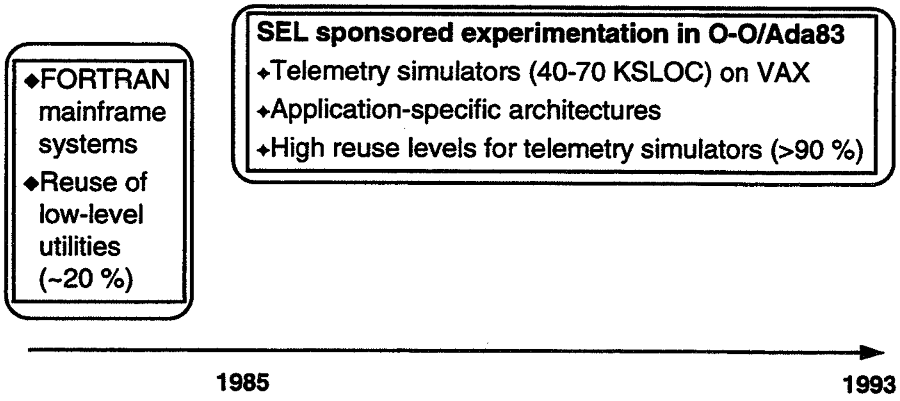


- FORTRAN mainframe systems
- Reuse of low-level utilities (~20 %)

1985

1993

Reuse History at FDD & SEL



- FORTRAN mainframe systems
- Reuse of low-level utilities (~20 %)

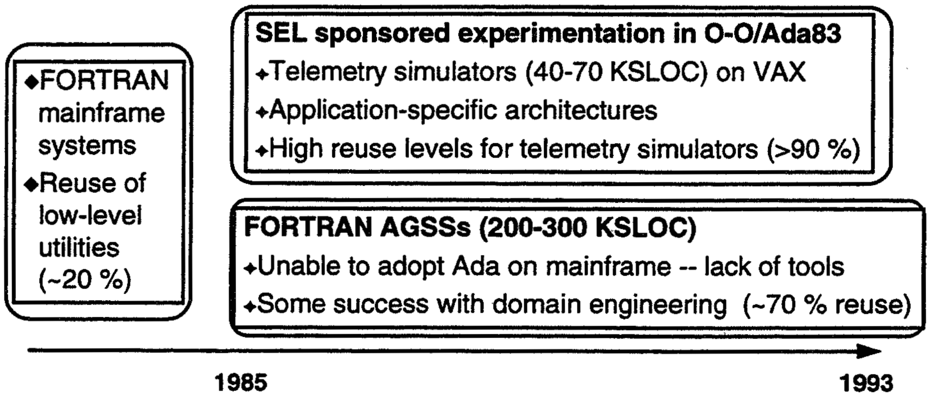
SEL sponsored experimentation in O-O/Ada83

- Telemetry simulators (40-70 KSLOC) on VAX
- Application-specific architectures
- High reuse levels for telemetry simulators (>90 %)

1985

1993

Reuse History at FDD & SEL

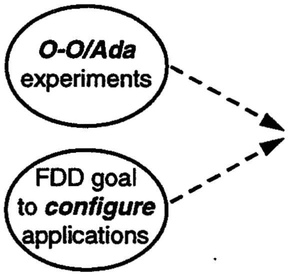


- FORTRAN mainframe systems
- Reuse of low-level utilities (~20 %)

SEL sponsored experimentation in O-O/Ada83
- Telemetry simulators (40-70 KSLOC) on VAX
- Application-specific architectures
- High reuse levels for telemetry simulators (>90 %)

FORTRAN AGSSs (200-300 KSLOC)
- Unable to adopt Ada on mainframe -- lack of tools
- Some success with domain engineering (~70 % reuse)

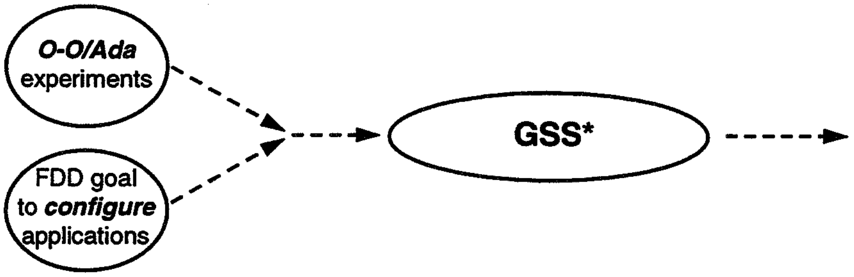
Evolution of GSS



O-O/Ada experiments

FDD goal to ***configure*** applications

Evolution of GSS



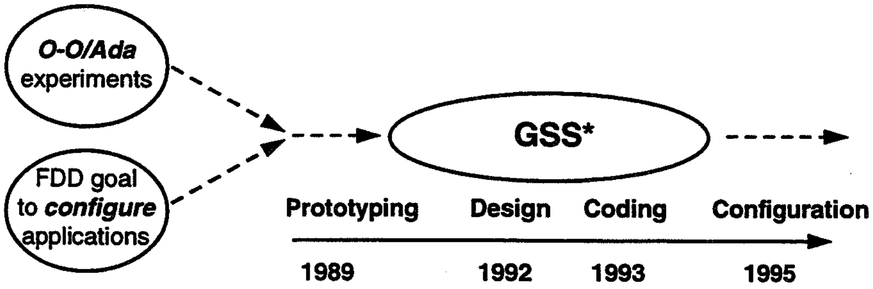
O-O/Ada experiments

FDD goal to ***configure*** applications

GSS*

***Generalized Support Software (GSS): a library of generalized, configurable application components developed with an object-oriented domain engineering approach.**

Evolution of GSS



O-O/Ada experiments

FDD goal to ***configure*** applications

GSS*

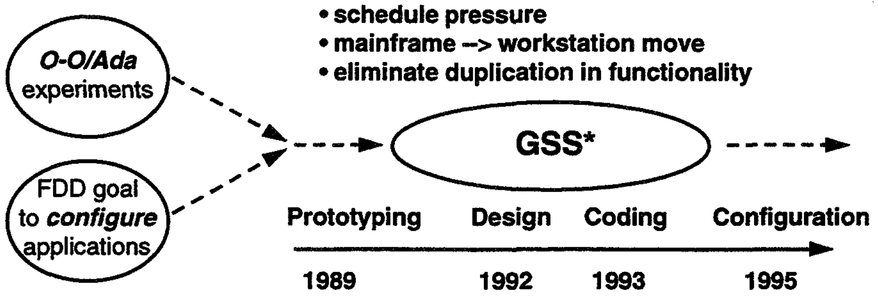
Prototyping	**Design**	**Coding**	**Configuration**
1989	**1992**	**1993**	**1995**

***Generalized Support Software (GSS): a library of generalized, configurable application components developed with an object-oriented domain engineering approach.**

Evolution of GSS

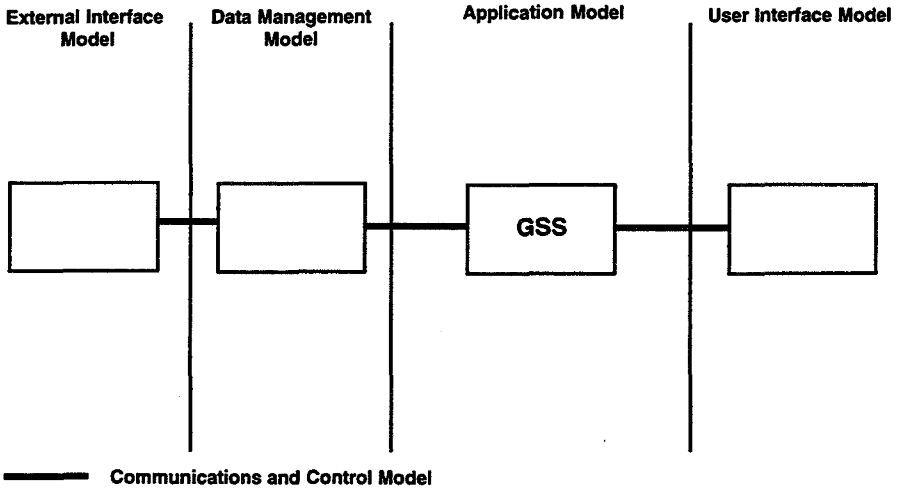
Pressures/Goals:

- **reduced budgets**
- **schedule pressure**
- **mainframe --> workstation move**
- **eliminate duplication in functionality**

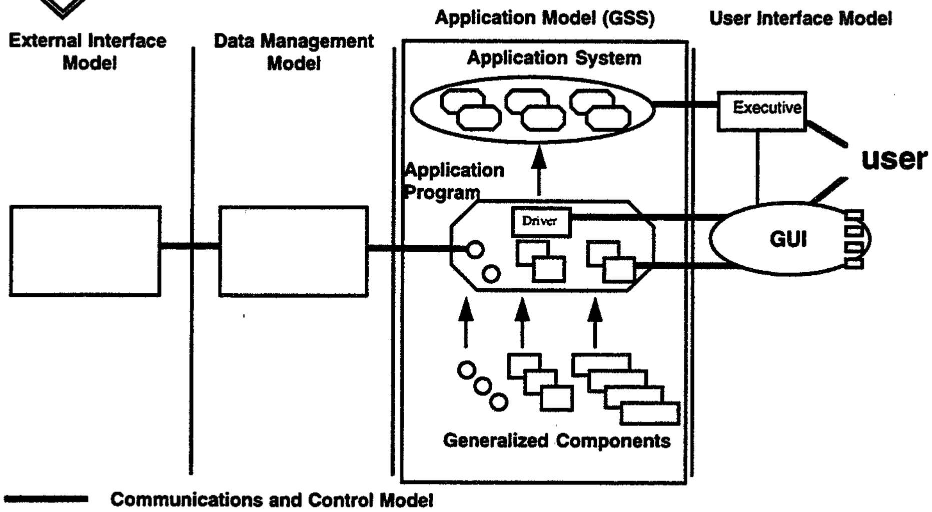


***Generalized Support Software (GSS): a library of generalized, configurable application components developed with an object-oriented domain engineering approach.**

FDDS/GSS Architecture



FDDS/GSS Architecture



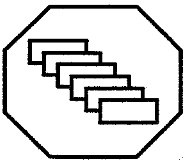
The GSS Architecture Hierarchy

Object: a model of some individual item of interest in the problem domain.

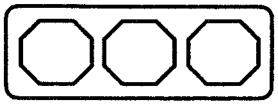
Applications

Reuse Library

Class: a generalized object



Category: a set of similar classes grouped together along with rules for using these member classes for mission support.



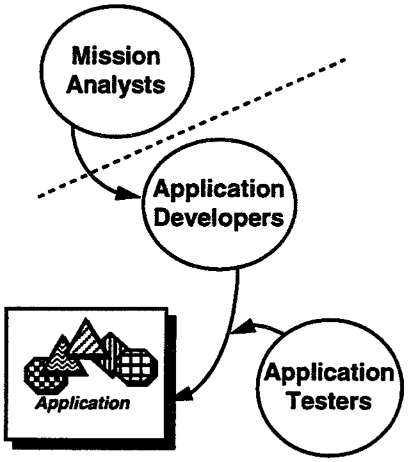
Subdomain: a group that contains all categories necessary to specify the functionality in a specific high-level area of the overall problem domain.

OOPSLA '96 – October 10, 1996

Project Organization

Experience Factory

Traditional **Experience Factory**



Mission Analysts

Application Developers

Application

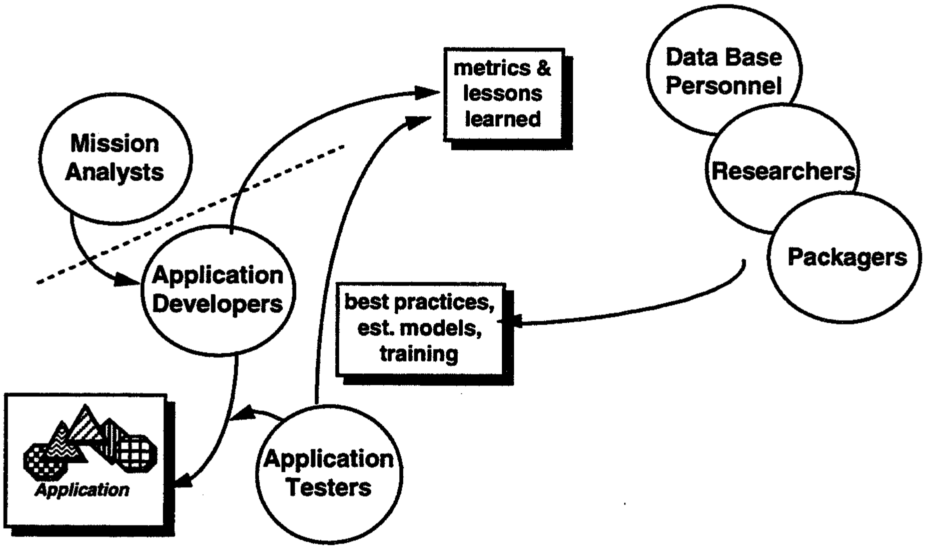
Application Testers

SEW21--December 4, 1996

Project Organization

Experience Factory

Traditional **Experience Factory**



metrics & lessons learned

Data Base Personnel

Researchers

Packagers

Mission Analysts

Application Developers

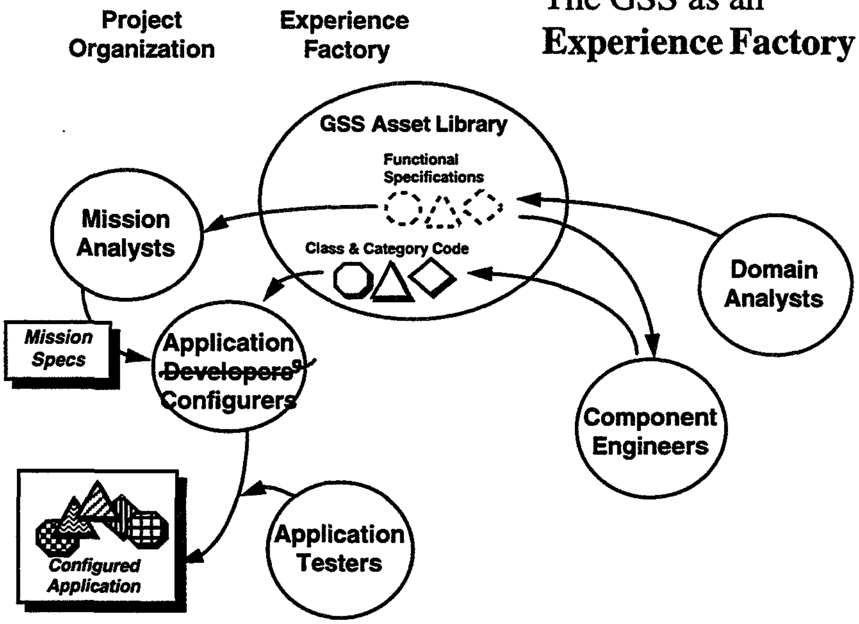
best practices, est. models, training

Application

Application Testers

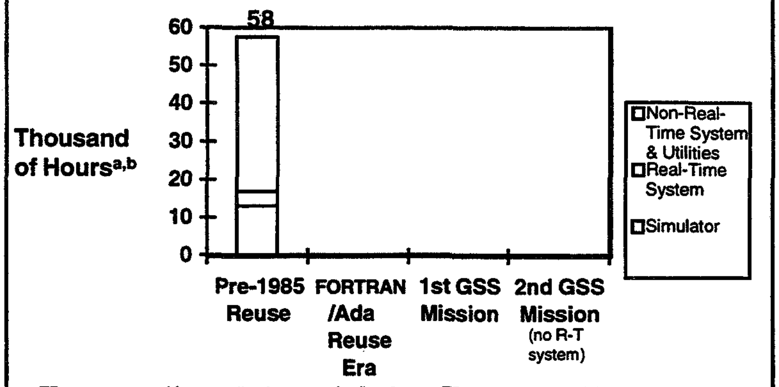
SEW21--December 4, 1996

The GSS as an **Experience Factory**



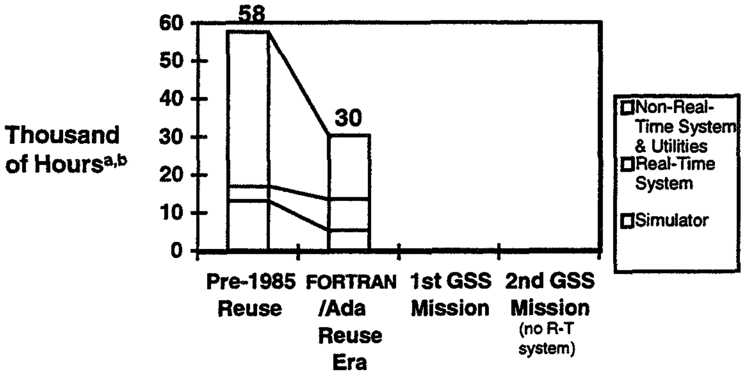
SEW21--December 4, 1996

GSS Reduces Deployment Costs



[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.
[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

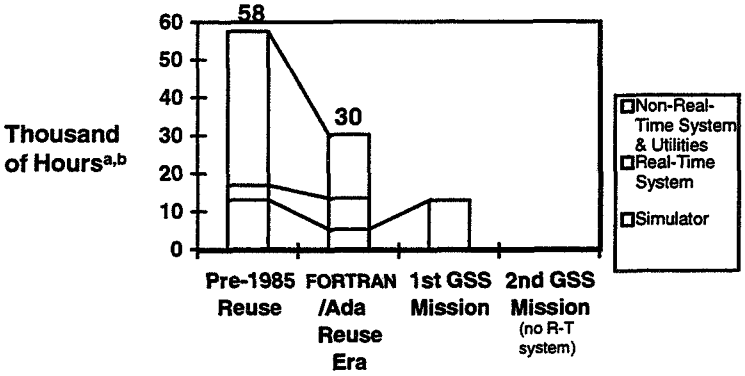
GSS Reduces Deployment Costs



[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.

[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

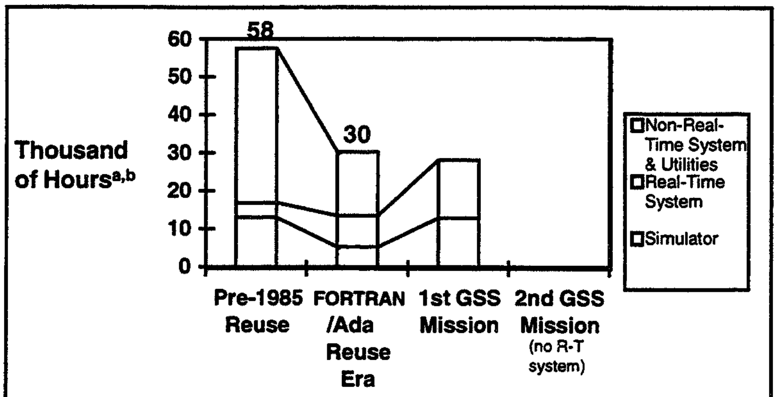
GSS Reduces Deployment Costs



[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.

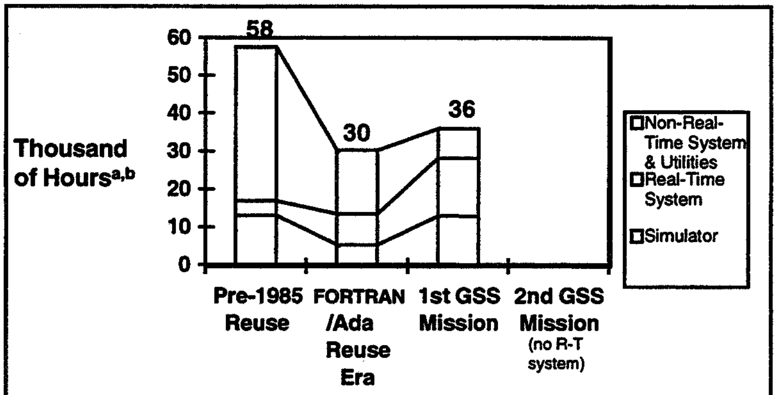
[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

GSS Reduces Deployment Costs



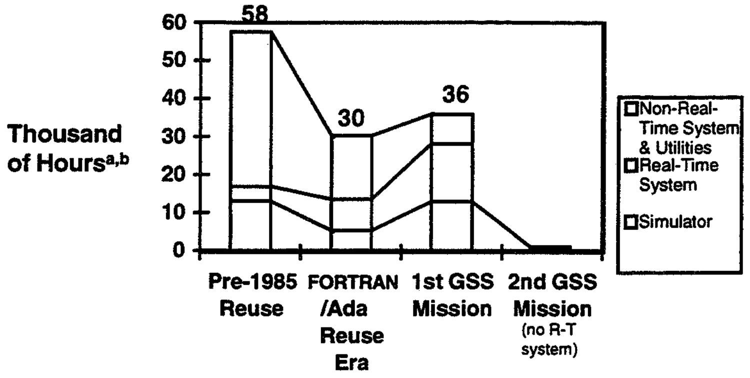
[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.
[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

GSS Reduces Deployment Costs



[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.
[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

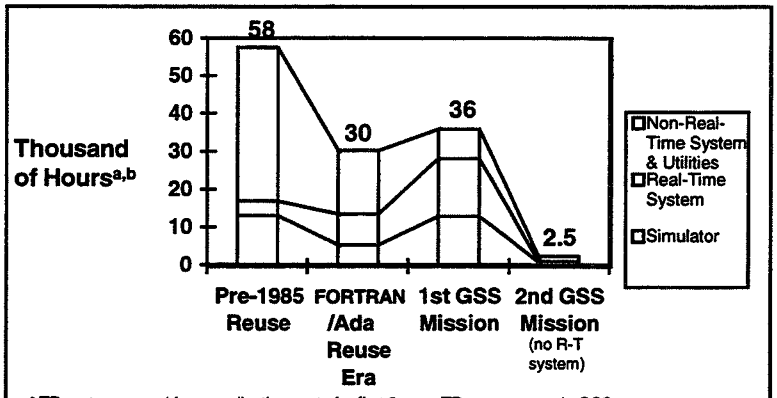
GSS Reduces Deployment Costs



[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.
[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

Most recent appliations cost on the order of 10% of pre-GSS costs.

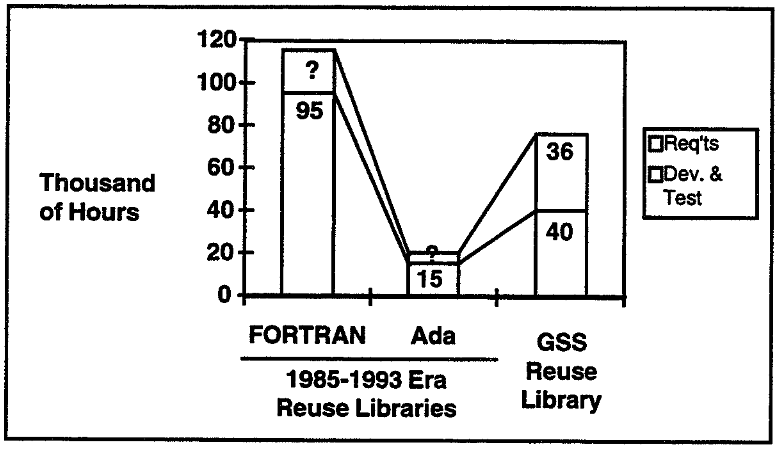
GSS Reduces Deployment Costs



[a] TP costs removed from application costs for first 2 eras; TPs unecessary in GSS era.
[b] Library maintenance costs included in 2nd era; GSS mission costs include total of 10 Khr of GSS overhead (library maintenance, etc.)

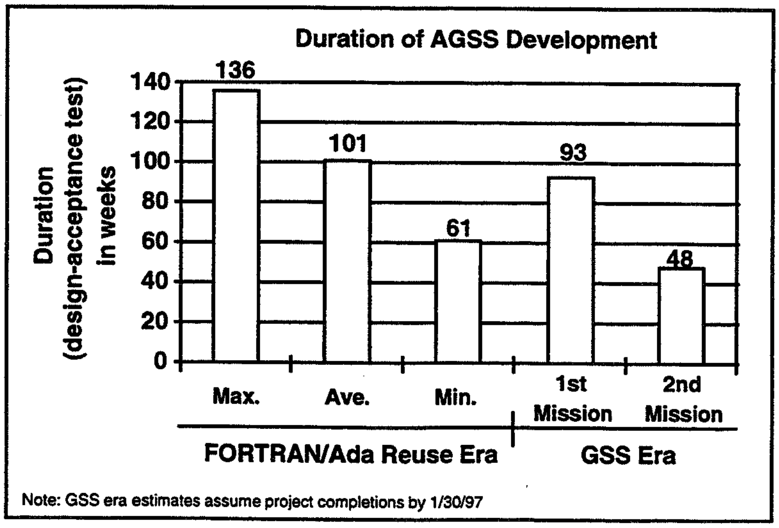
Most recent applications cost on the order of 10% of pre-GSS costs.

Library Investment Cost



Deployment savings likely to recoup GSS investment by 4th mission.

GSS Reduces Deployment Cycle Time



Note: GSS era estimates assume project completions by 1/30/97

Issues with the GSS Process

- GSS viewed as a "child" of the S/W developers.
- Can't write the (GSS) mission spec without understanding the GSS functional specs.
- The GSS functional specs (1600 pages) are written by and for developers -- not for analysts.
- Very few analysts involved in GSS process.
- Many analysts cool towards GSS.

Potential Improvements for the GSS Process

- Create a database for mission requirements (text-based now) in order to reduce mission spec effort.
- Automate the generation of mission specifications and configuration inputs.
- Create a scenario-driven overlay -- designed by analysts -- for the functional specs.

Evolving Techologies

- O-O languages evolving: Ada83 --> C++ and Ada95
  - GSS Attitude Subdomain in Ada83
  - GSS Mission Planning Subdomain in C++
- O-O design techniques evolving
  - use cases (scenario-driven)
- Marketplace GUI's more advanced now
- COTS products more powerful, more varied

Alternative Reuse Processes Now Available

- FORTRAN reuse libraries were rehosted to workstations using COTS products;
  can support future missions as well.
- Other COTS products being used for mission support.
- New missions can choose GSS and/or COTS.

Understanding Alternative Reuse Processes

- Would GSS benefit from a different GUI?
- Does O-O Tech. in GSS make it more robust or maintainable than non-O-O COTS products?
- Other maintenance issues
- Performance
- Reliability
- Portability
- Documentation

Conclusions

- GSS process savings
  - Deployment time.
  - Application deployment costs --> 10% of pre-GSS costs.
  - Recoup library investment in 4 missions?
- GSS not designed for FDD analysts
  - Functional specs, mission specs, configuration process
  - Mods needed to make GSS process more useful to analysts.
- Alternative reuse processes now available.
- More work needed to compare and assess GSS and COTS.

S3-61
66446
366695
P. 16

Studies on Reading Techniques

Victor Basili, Gianluigi Caldiera, Filippo Lanubile, and Forrest Shull

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, MD, USA
{basili | gcaldiera | lanubile | fshull}@cs.umd.edu

1. Introduction

Software reading is a key technical activity that aims at achieving whatever degree of understanding is needed to accomplish a particular objective. The various work documents associated with software development (e.g., requirements, design, code, and test plans) often require continual understanding, review and modification throughout the development life cycle. Thus software reading, i.e., the individual analysis of textual software work products, is the core activity in many software engineering tasks: verification and validation, maintenance, evolution, and reuse.

Through our work in the SEL, we have evolved our understanding of reading technologies via experimentation. We have run empirical studies ranging from blocked subject-project experiments (reading by step-wise abstraction vs. functional and structural testing [Basili,Selby87]) to replicated projects (University of Maryland Cleanroom study [Selby,Basili,Baker87]) to a case study (the first SEL Cleanroom study) to multi-project variation (the set of SEL Cleanroom projects [Basili,Green94]) and most recently, back to blocked subject-project experiments (scenario-based reading vs. current reading [Basili,Green, Laitenberger,Lanubile,Shull,Soerumgaard,Zelkowitz96], [Porter,Votta,Basili95]).

We have used a variety of experimental designs to provide insight into the effects of different variables on reading. The experiments are based upon the ideas that reading is a key technical activity for improving the analysis of all kinds of software documents and that we need to better understand its effect. We believe these studies demonstrate the evolution of knowledge about reading, experimentation, and the packaging of experimental results over time. Several of these experiments have been replicated by other researchers.

To provide a technological base to software reading, we attempt to develop specific reading techniques, made up of a concrete set of instructions which are given to the reader on how to read or what to look for in a software work product. Our current research efforts focus on the development of families of reading techniques, based on empirical evaluation. Each family of reading techniques can be parameterized for use in different contexts and must be evaluated for those contexts.

The taxonomy of reading families is shown in Figure 1. The upper part of the tree (over the dashed horizontal line) models the problems that can be addressed by reading. Each level represents a further specialization of the problem according to classification attributes which are shown in the rightmost column of the figure. For example, reading (*technology*) can be applied for analysis (*high level goal*), more specifically to detect faults (*specific goal*) in a requirements specification (*document*) which are written in English (*notation/form*).

The lower part of the tree, (below the dashed horizontal line) models the specific solutions we have provided to date for the particular problems, represented by each path down the tree. The solution space consists of reading families and reading techniques. Each family is associated with a particular goal, document or software artifact, and notation in which the document is written. Each technique within the family is: (1) tailorable, based upon the project and environment characteristics; (2) detailed, in that it provides the reader with a well-defined set of steps to follow; (3) specific, in that the reader has a particular purpose or goal for reading the document and the procedures that support the goal; (4) focused, in that it provides a particular coverage of the document, and a combination of techniques in the family provides coverage of the entire document. Finally each technique is being studied empirically to determine if and when it is most effective.

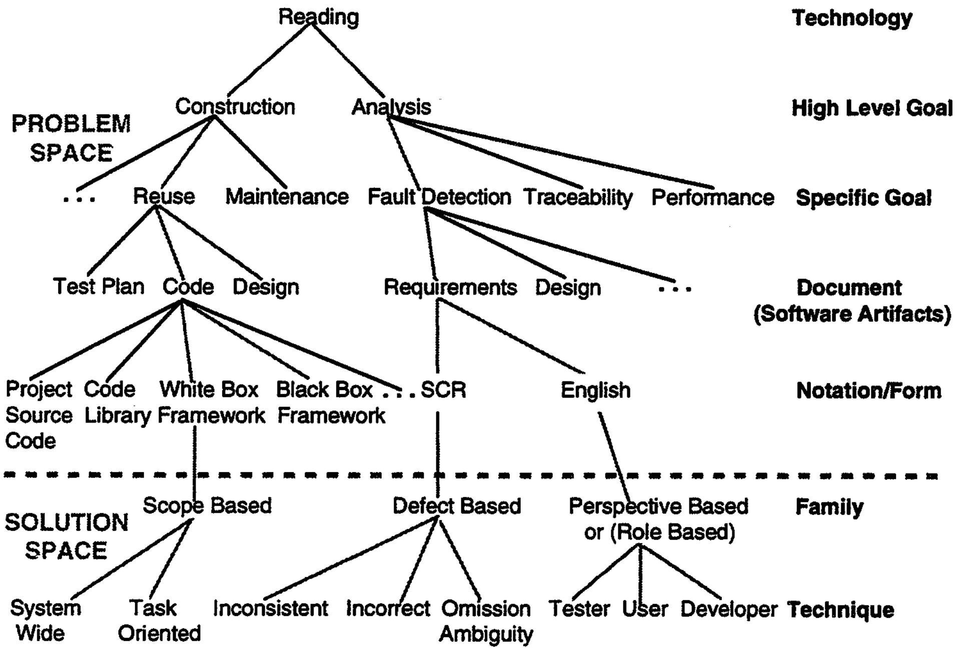


Figure 1. Families of reading techniques

Each software life cycle phase contains both construction and analysis activities. The design phase, for example, is responsible for creating design documents, as well as for analyzing their quality. Since construction and analysis are two parts of the same phase, you can learn from analysis technologies about construction technologies. At a high level, we divide reading activities into Reading for Analysis and Reading for Construction, to parallel this distinction between analysis and construction processes and to show that the usefulness of good reading techniques is not limited to any narrow portion of the software life-cycle. The next two sections describe our work in these areas.

2. Reading for Analysis

Reading for analysis is aimed at answering the following question: Given a document, how do I assess various qualities and characteristics? Reading for analysis is important for product quality; it can help us understand the types of defects we make, and the nature and structure of the product. It can be used for various documents throughout the life cycle. Besides helping us assess quality, it can provide insights into better development techniques.

Our research into reading for analysis has so far emphasized *defect detection*; we have focused on the requirements phase for this purpose. We have generated two families of reading techniques (collectively known as *scenario-based reading*), by creating operational scenarios which require the reader to first create an abstraction of the product, and then answer questions based on analyzing the abstraction with a particular emphasis or role that the reader assumes. Each reading technique in a family can be based upon a different abstraction and question set. The choice of abstraction and the types of questions may depend on the document being read, the problem history of the organization, or the goals of the organization.

The first family of scenario-based reading techniques is known as *defect-based reading*, and focuses on a model of the requirements using a state machine notation. The different model views are based upon focusing on specific defect classes: data type inconsistency, incorrect functions, and ambiguity or missing information. The analysis questions are generated by combining and abstracting a set of questions that are used in checklists for evaluating the correctness and reliability of requirements documents.

The second family of techniques, *perspective-based reading*, focuses on different product perspectives, e.g., reading from the perspective of the software designer, the tester, the end-user, the maintainer, the hardware engineer. The analysis questions are generated by focusing predominantly on various types of requirements errors (e.g., incorrect fact, omission, ambiguity, and inconsistency) by developing questions that can be used to discover those errors from the one perspective assumed by the reader of the document (e.g., the questions for the tester perspective lead the reader to discover those requirement errors that could be found by testing the final product).

In order to understand the effectiveness of scenario-based reading techniques in particular, we have experimentally studied techniques from both families. The first series of experiments [Porter,Votta,Basili95], [Basili, Green, Laitenberger, Lanubile, Shull, Soerumgaard, Zelkowitz96] was aimed at discovering if scenario-based reading is more effective than current practices. Based upon these experiments, we had empirical evidence that scenario-based reading techniques can improve the effectiveness of reading methods. At the same time, we noted that some scenarios were less effective than others. We give some details of these experiments here in order to illustrate our own experiences with experimentation in software engineering.

2.1 Defect-Based Reading Experiment

In the defect-based reading study [Porter,Votta,Basili95], we evaluated and compared defect-based reading, ad hoc reading and checklist-based reading, with respect to their effect on fault detection effectiveness in the context of an inspection team. The study, a blocked subject-project, was replicated twice in the spring and fall of '93 using 48 graduate students at the University of Maryland. The experimental design was a partial fractional factorial design. The design was less elegant than the [Basili,Selby87] design because the comparison here is with the status quo approach (ad hoc) or with a less procedurally organized approach (checklists) so it is impossible to teach the subject a defect-based reading approach and then return to ad hoc or check list. In this case, a sort of ordering was assumed. On the first pass there were more ad hoc and check list readers. Several, but not all, were moved to defect-based reading on the second pass.

Major results were that:

- the defect-based readers performed significantly better than ad hoc and checklist readers;
- the defect-based reading procedures helped reviewers focus on specific fault classes but were no less effective at detecting other faults; and
- checklist reading was no more effective than ad hoc reading.

2.2 Perspective-Based Reading Experiment

In the perspective-based reading study [Basili, Green, Laitenberger, Lanubile, Shull, Soerumgaard, Zelkowitz96], we evaluated and compared perspective-based reading and NASA's current reading technique with respect to their effect on fault detection effectiveness in the context of an inspection team. Three types of perspective-based reading techniques were defined and studied: tester-based, designer-based, and user-based. The study, again a blocked subject-project, was run twice in the SEL environment with NASA professionals.

The design evaluated the effectiveness of perspective-based reading on both domain-specific and generic requirements documents, which had been constructed expressly so that the generic portion could be replicated in a number of different environments, while the domain-specific part could be replaced in each new environment. This would allow us to combine the generic parts from multiple studies but focus on improvement local to a particular environment. Based on feedback from the subjects and other difficulties encountered in the first run of the experiment, we were able to make changes to the experimental design that improved the second run. For example, we found it necessary to:

- Include more training sessions, to make certain that subjects were familiar with both the documents and techniques involved in the experiment;

- Allow less time for each review of the document, since subjects tended to tire in longer sessions;
- Shorten some of the documents, to reach a size that could realistically be expected to be checked in an experimental, time-constrained setting.

Major results of this experiment were that:

- both team and individual scores improved when perspective-based reading was applied to generic documents
- team scores improved when perspective-based reading was applied to NASA documents

Although the true benefit of PBR is expected to be seen at the level of teams which combine several different perspectives for improved coverage, the results for individuals showed that the use of PBR may lead to improvements at the individual level as well. Thus, we further analyzed the individual reviewers' performance with the generic documents considering other attributes of effectiveness. Preliminary results of this second-round analysis were that:

- PBR reviewers took more time than reviewers using their current reading technique but the average cost for finding a defect was the same for both the methods
- The percentage of false positives for both methods is about the same. There were less false positives with PBR although the difference was not significant)

If we consider that PBR reviewers found more defects than reviewers using their current reading technique and assume that the cost of finding a defect increases as more defects are found, we can conclude (for generic documents) that:

- PBR is actually more productive than the local reading technique.
- The relative effort spent fixing defects is better for PBR.

By tailoring the perspectives also to the NASA application domain, we should be able to improve individual performance on these tasks. We need to improve the treatments used in the reading techniques. This can be done by developing questions for each scenario using the specific application domain (e.g., flight dynamics requirements documents), by focusing on the abstraction mechanism used (e.g., using a specific technique like equivalence partition testing for the testing perspective), or focusing the questions to cover certain classes of defects more effectively.

We need to add a qualitative component to the controlled studies to gather more insights into what is needed to better set up the experiment, define the terminology, and interpret the results. For example, controlled experiments could be supplemented with various standard methods in qualitative analysis such as the use of pre-tests, post-tests, ethnographic studies, and interviews.

3. Reading for Construction

Reading for construction is aimed at answering the question: Given an existing system, how do I understand how to use it as part of my new system? Reading for construction is important for comprehending what a system does, what capabilities exist and do not exist; it helps us abstract the important information in the system. It is useful for maintenance as well as for building new systems from reusable components and architectures.

Our emphasis here has so far focused on the *reuse* of an existing system or library. Reusing class libraries does increase quality and productivity, but class libraries do not provide default system behavior but only functionality at a low level, and force the developer to provide the interconnections between the libraries. Greater benefits can be expected from reusable, domain specific architectures and components that are of sufficient size to be worth reusing. Thus, we are currently focusing on the reuse allowed by object-oriented frameworks for this purpose [Lewis95].

Since a framework provides a pre-defined class hierarchy, object interaction, and thread of control, developers must fit their applications into the framework. This means that in framework-based development, the static structure and dynamic behavior of the framework must first be understood and then adapted to the specific requirements of the application. It is assumed that the effort to learn the framework and develop code within the system is less than the effort required to develop a similar system from scratch. Although it is recognized that the effort required to learn enough about the framework to begin coding is high [Booch94], [Pree95], [Taligent95], little work has been done in the way of minimizing this learning curve.

3.1 White-Box Frameworks

We are studying the process of learning such a framework (or more generally, any unfamiliar system) and developing constructive reading techniques that may minimize the effort expended on program understanding in particular situations. This experiment involves the study of a *white-box framework*, which defines a set of interacting classes, usually abstract classes, that capture the invariants in the problem domain. Since the source code of the classes is accessible to the programmer, a white-box framework can be specialized by deriving application-specific classes from the base classes through inheritance and by completing or overriding their methods [Johnson,Foote88], [Schmid96]. Learning to use a white-box framework is the same as learning how it is constructed because the user must have detailed framework code knowledge.

We have defined two reading techniques for using a white-box framework to build new applications: a system-wide reading technique and a task-oriented reading technique. Both techniques look at the static structure and the run-time behavior of the framework, and both have access to the same sources of information. The main difference is the focus of the learning process: the system-wide technique focuses more on the big picture than on the detailed task to be accomplished (which is the focus of the task-oriented technique).

With the system-wide reading technique, programmers attempt to gain a broad knowledge of the framework design. As a consequence, they deliver the functionality required by the new application mainly by specializing the abstract classes of the framework. With the task-oriented reading technique, programmers use existing framework-based applications as examples and attempt to gain a specialized knowledge of the parts which are directly relevant for the required system. As a consequence, they deliver the functionality required by the new application mainly by changing the concrete classes of the examples.

To compare these two techniques we have conducted a repeated-project experiment, in which we present graduate students and upper-level undergraduates with an application task to be developed using the white-box framework ET++ [Lewis95]. ET++ is a sophisticated framework that poses learning problems which can be major inhibitors against its use. The overall goal of the experiment is to compare the reading techniques for framework understanding (system-wide and task-oriented) with respect to their effect on ease of framework learning and usage, i.e., the ease with which the framework is understood and functionality is added. Students receive separate lectures on the reading techniques and work in teams of three people. One half of the class has been taught the system-wide reading technique and the other half the task-oriented reading technique. Preliminary results show that

- Even a relatively well-designed although poorly documented framework presents many difficulties in learning how to derive framework-based applications
- Students demonstrated an overhead in learning the framework with high levels of frustration in the early weeks because of the investment in time without an immediate payoff in programming
- Students found it easier to learn in the beginning by reading and reusing example applications than by trying to first gain a comprehensive knowledge of the framework
- Difficulties were encountered with the system-wide technique because the documentation provided was at an insufficient level of detail to be useful, and because the technique gave little guidance as to which area of the framework to concentrate on first.
- Difficulties were encountered with the task-oriented technique because it was hard to find suitable examples for all required functionality and because example applications were sometime inconsistent in terms of structure and organization.

3.2 Black-Box Frameworks

Black-box frameworks allow an application to be created by composing objects rather than by programming [Johnson, Foote88], [Schmid96]. They provide alternative concrete classes which have to be selected when creating an application, allowing some variability in the applications created. Thus, a black-box framework is customized by selecting, parameterizing, and configuring a set of components that provide the application specific behavior.

The interface between components can be defined by protocol, so the user needs to understand only the external interface of the components. Since this does not require knowledge of the framework code, black-box frameworks could be considered easier to use than white-box frameworks. However, better documentation and training are required because developers cannot look at the source code to determine what is going on.

We intend to investigate reading techniques for black-box frameworks in a real development context, focusing on the Generalized Support Software (GSS), a black-box framework developed and used to enable much more rapid deployment of flight dynamics applications at NASA/GSFC. The process for configuring a new mission-support application with GSS consists of selecting GSS classes to compile and link together, and setting values for a large number of control and operational parameters. The size and sophistication of the reuse asset library poses learning problems which can be major inhibitors against its use. Here, the goal is to improve the existing reading techniques which are used to understand which generalized components must be configured in order to develop new applications.

Variations of the two reading techniques that were compared in the ET++ experiment (system-wide and task-oriented) will have to be designed for use with a black box framework. The idea behind each reading technique will be the same, however. One will require the framework user to learn the overall structure of the framework, while the other will help the student learn with specific examples.

4. Conclusions

Much of our work in reading has so far focused on three families of reading techniques:

1. the defect-based reading family for analyzing requirements specification written in SCR notation, with the purpose of defect detection;
2. the perspective-based reading family for analyzing requirements specification written in English language, with the purpose of defect detection;
3. the scope-based reading family for constructing applications through reuse of white-box frameworks.

We will continue to conduct empirical studies which will allow us to closely monitor the use of different reading techniques in laboratory and real projects, both quantitatively and qualitatively. We believe it is necessary to integrate results from both types of studies in order to gain a deeper understanding of the research questions.

As our ability to understand software reading as a technique evolves, we plan to develop other families of reading techniques parameterized for use in different contexts and empirically evaluated for those contexts..

References

V. Basili, S. Green, “Software process evolution at the SEL”, *IEEE Software*, pp.58-66, July 1994.

V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Soerumgaard, M. Zelkowitz, “The empirical investigation of perspective-based reading”; *Empirical Software Engineering - An International Journal*, vol. 1, no. 2, 1996.

V. Basili, R. Selby, “Comparing the effectiveness of software testing strategies”, *IEEE Transactions on Software Engineering*, vol. SE-13, no. 12, pp.1278-1296, December 1987.

G. Booch, , “Designing an application framework”, *Dr. Dobb's Journal*, vol. 19, no. 2, February 1994.

R. Johnson, B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, June 1988

T. Lewis et al., *Object-Oriented Application Frameworks*, Manning Publications Co., 1995.

A. Porter, L. Votta, V. Basili, "Comparing detection methods for software requirements inspections: a replicated experiment", *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp.563-575, 1995.

W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Publishing Co., 1995.

H. Schmid, "Creating Applications from Components: A Manufacturing Framework Design", *IEEE Software*, vol. 13, no. 6, pp. 67-75, November 1996.

R. Selby, V. Basili, Baker, "Cleanroom software development: an empirical evaluation", *IEEE Transactions on Software Engineering*, pp.1027-1037, September 1987.

Taligent Inc., *The Power of Frameworks*, Addison-Wesley, 1995.

Studies on Reading Techniques

Victor R. Basili, Gianluigi Caldiera,
Filippo Lanubile, Forrest Shull

**Experimental Software Engineering Group (ESEG)
University of Maryland, College Park**

1996

Reading Motivation

Reading is a key technical activity
for analyzing and constructing software documents

We need to evolve reading technology
by improving the analysis of all kinds of software documents

What is software reading?
the individual analysis of a textual software product
e.g., requirements, design, code, test plans
to achieve the understanding needed for a particular task
e.g., defect detection, reuse, maintenance

We have evolved our understanding of reading technology in the SEL
via a series of experiments
from the early reading vs. testing experiments
to various Cleanroom experiments
to the development of new reading techniques currently under study

Reading Research

What is a reading technique?

a concrete set of instructions given to the reader
saying how to read and what to look for in a software product

Our current research efforts are to

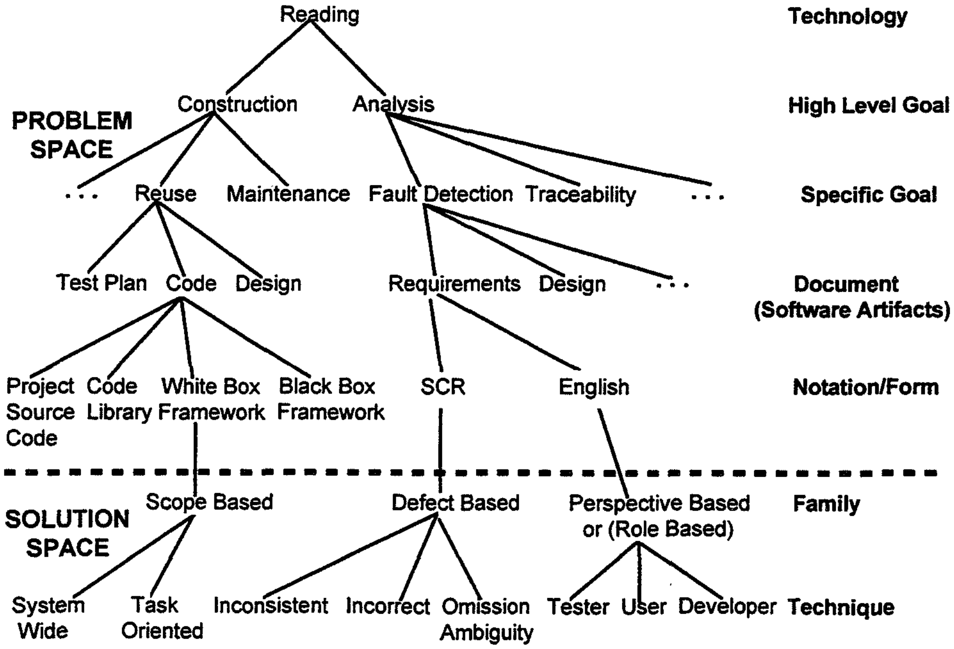
develop families of reading techniques
based on empirical evaluation
parameterized for use in different contexts
evaluated for those contexts

In this talk we discuss

a taxonomy of reading families
specific techniques and experimental evaluations
where we are going in our research program

V.R. BASILI SEL-21 1996

Families of Reading Techniques



Reading — **Technology**

Construction, Analysis — **High Level Goal**

PROBLEM SPACE

. . . Reuse Maintenance Fault Detection Traceability . . . — **Specific Goal**

Test Plan Code Design Requirements Design . . . — **Document (Software Artifacts)**

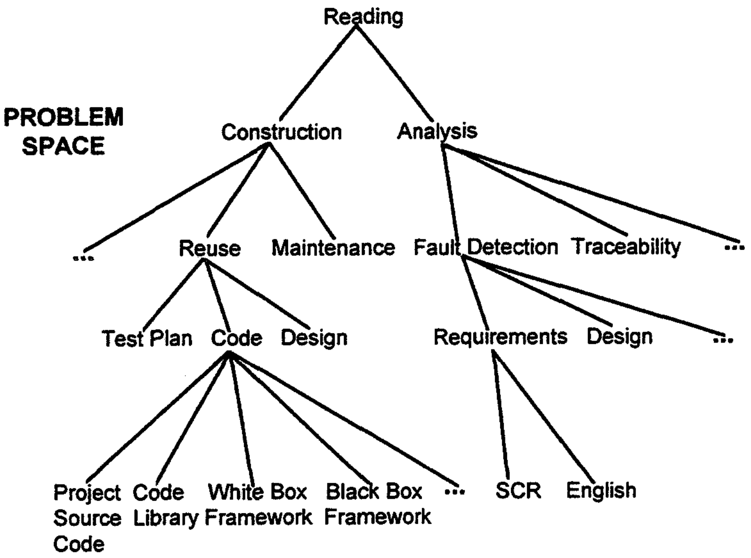
Project Source Code, Code Library, White Box Framework, Black Box Framework, SCR, English — **Notation/Form**

SOLUTION SPACE

Scope Based, Defect Based, Perspective Based or (Role Based) — **Family**

System Wide, Task Oriented, Inconsistent, Incorrect, Omission, Ambiguity, Tester, User, Developer — **Technique**

Families of Reading Techniques



High Level Reading Goals

We differentiate two goals for reading techniques:

Reading for analysis:
Given a document,
how do I assess
various qualities
and characteristics?

Assess for
product quality
defect detection
...

Useful for
quality control,
insights into development
...

Reading for construction:
Given a system,
how do I understand
how to use it as part
of my new system?

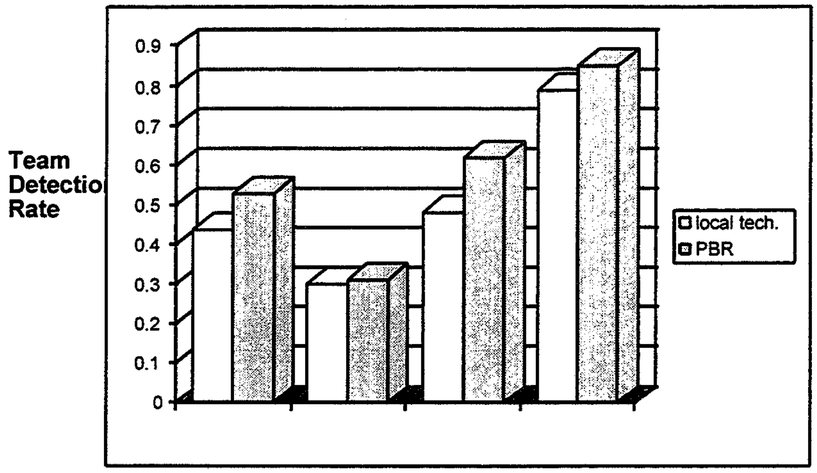
Understand
what a system does
what capabilities do and do not exist
...

Useful for
maintenance
building systems from reuse
...

Reading for Analysis: Perspective-Based Reading Experiment

Goal of Perspective-Based Reading (PBR):
detect defects in a requirements document
focus on <u>product consumers</u>

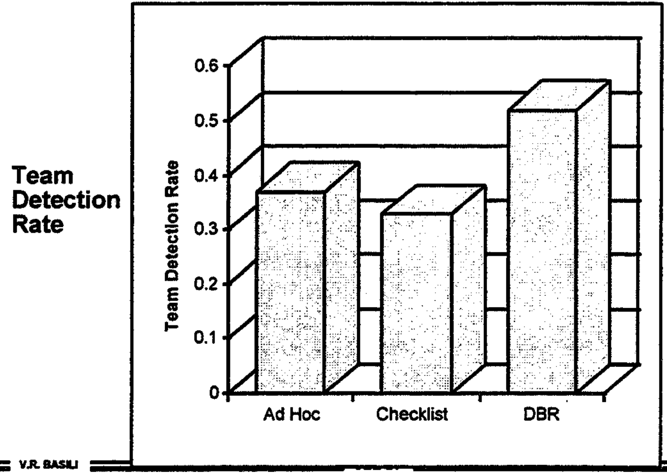
Controlled experiment run twice with NASA professionals:



Reading for Analysis: Defect-Based Reading Experiment

Goal of Defect-Based Reading (DBR):
detect defects in a requirements document
focus on <u>defect classes</u>

Controlled experiment run twice with UMD graduate students:



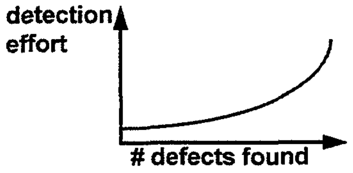
Experiments with Reading for Analysis

More Results from the PBR Analysis

Generic Domain at the Individual Level:

PBR **found more defects** than the local Reading Technique
PBR **took more time** than the local Reading Technique
And the **average cost for finding a defect** is the same for both methods

Assuming that cost of finding a defect increases as more defects are found



Might imply: PBR is more productive than the local Reading Technique

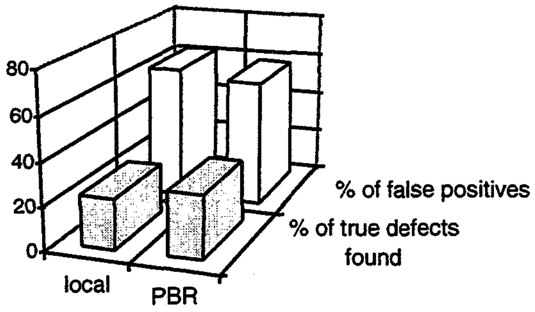
V.R. BASILI SEL-21 1996

Experiments with Reading for Analysis

More Results from the PBR Analysis

Generic Domain at the Individual Level:

PBR **found more defects** than the local Reading Technique
The percentage of false positives for both methods is about the same



Might imply: Relative effort spent fixing defects later is better for PBR

V.R. BASILI SEL-21 1996

Reading for Construction

Interested in reading techniques
to minimize the effort to learn a new tool or existing system for a specific application development

Framework

A set of classes augmented with a built-in model for defining how classes interact
- to reuse domain concepts
- to encapsulate implementation details

Framework
(domain specific)
Custom Software
(application specific)

Two approaches:
- **White-box frameworks** - extend and modify classes
- **Black-box frameworks** - select and configure ready-made classes

Experiments with Reading for Construction

White-Box Frameworks

We proposed two reading techniques emphasizing different facets of the framework:

System-wide technique:
- study classes
- gain a broad knowledge of the framework design
- build system by choosing appropriate classes

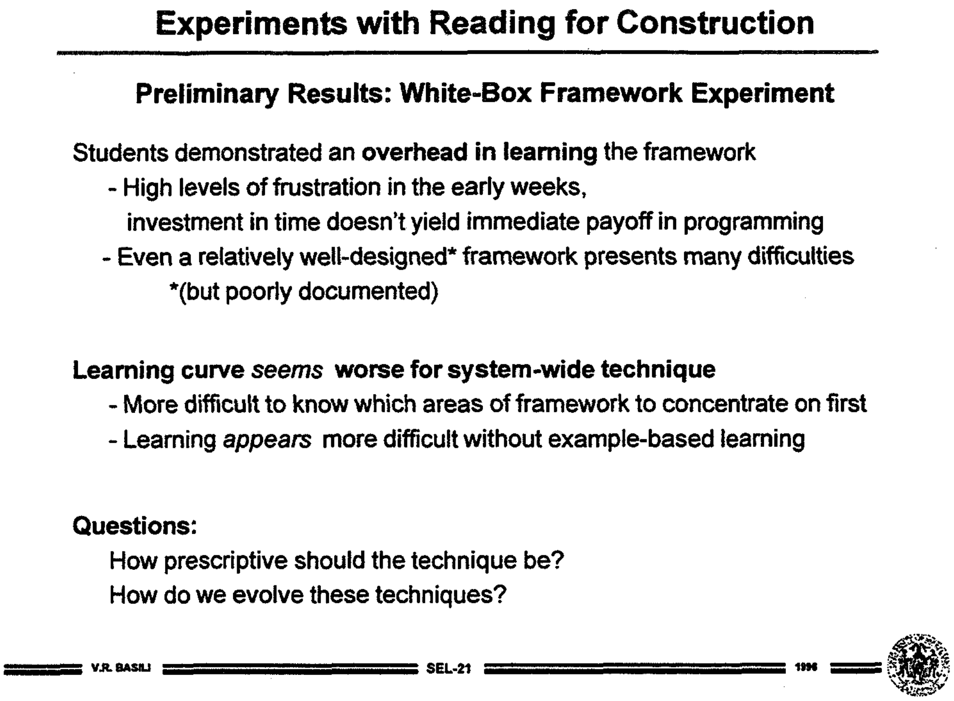
Task-oriented technique:
- study examples
- gain a specialized knowledge of directly relevant system parts
- build system by modifying examples

Experimental design:

Repeated project - 45 subjects - 15 three person teams

Environment:

University of Maryland upper-level software engineering course
Project: developing an OMT diagram editor - GUI framework ET++



Experiments with Reading for Construction

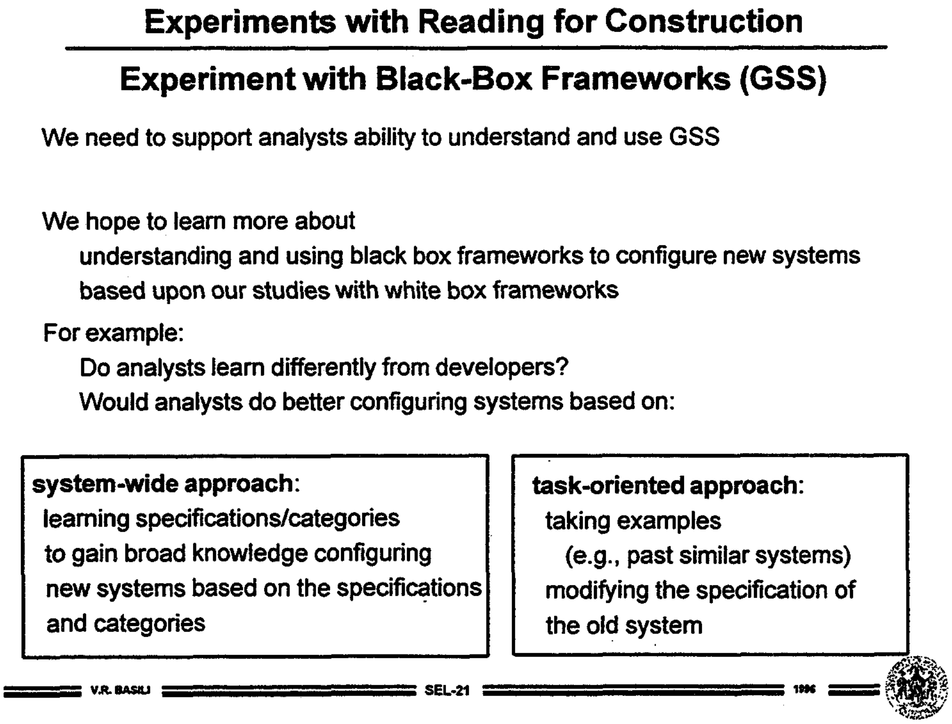
Preliminary Results: White-Box Framework Experiment

Students demonstrated an **overhead in learning** the framework
- High levels of frustration in the early weeks,
  investment in time doesn't yield immediate payoff in programming
- Even a relatively well-designed* framework presents many difficulties
  *(but poorly documented)

Learning curve *seems* worse for system-wide technique
- More difficult to know which areas of framework to concentrate on first
- Learning *appears* more difficult without example-based learning

Questions:
How prescriptive should the technique be?
How do we evolve these techniques?

V.R. BASILI SEL-21 1996



Experiments with Reading for Construction

Experiment with Black-Box Frameworks (GSS)

We need to support analysts ability to understand and use GSS

We hope to learn more about
understanding and using black box frameworks to configure new systems
based upon our studies with white box frameworks

For example:
Do analysts learn differently from developers?
Would analysts do better configuring systems based on:

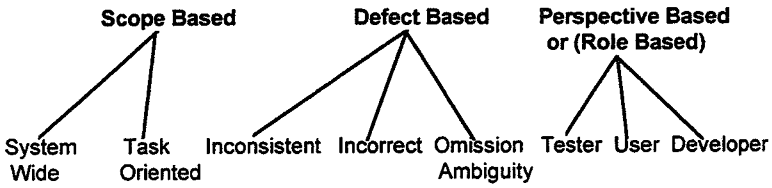
system-wide approach:
learning specifications/categories
to gain broad knowledge configuring
new systems based on the specifications
and categories

task-oriented approach:
taking examples
(e.g., past similar systems)
modifying the specification of
the old system

V.R. BASILI SEL-21 1996

Conclusion

We have developed three families of reading techniques
parameterized for use in different contexts and
evaluated experimentally in those contexts



Scope Based

System Wide

Task Oriented

Defect Based

Inconsistent

Incorrect

Omission

Ambiguity

Perspective Based or (Role Based)

Tester

User

Developer

V.R. BASILI SEL-21 1996

Long Range Research Plan

We need to

Develop better **empirical evaluation methods** to study these techniques in the laboratory and in industrial settings

Provide an **Experience Base** of technology evaluations that can be added to by other researchers and practitioners based upon their experiences with the technologies

Develop other **families of reading techniques**

and then

Develop **families of other techniques**
based on empirical evaluation
parameterized for use in different contexts
evaluated for those contexts

V.R. BASILI SEL-21 1996

OMIT THIS
PAGE

Session 2: Process

Software Development Technology Evaluation: Proving Fitness-for-Use with Architectural Styles
J. Cusick and W. Tepfenhart, AT&T

Systematic Process Improvement in a Multi-Site Software Development Project
H. Hientz, G. Smith, A. Gustavsson, P. Isacsson and C. Mattsson, Q-Labs GmbH

An Empirical Study of Process Conformance
S. Sorumgard, Norwegian University of Science and Technology

AT&T

AT&T Operations & Service Management

S4-61
415647
360696
P 22

Software Development Technology Evaluation: Proving Fitness-for-Use with Architectural Styles

21st NASA SEL Software Engineering Workshop
December 4-5, 1996
Greenbelt, MD

James Cusick
AT&T
Bridgewater, NJ
James.Cusick@att.com

William M. Tepfenhart
AT&T
Middletown, NJ
William.Tepfenhart@att.com

Software Development Technology Evaluation: Proving Fitness-for-Use with Architectural Styles

1. OVERVIEW

A cursory glance at a few trade journals will indicate that hundreds if not thousands of development tools are available on the market. Today, with the boom in Internet technologies, dozens of new tools enter the market place each month. Faced with this situation we were asked to define how to choose the best tools for use in the development of hundreds of AT&T's business applications. Starting in early 1995 we began a revitalization of the software tool assessment practices of AT&T and especially AT&T's Network Services Division (NSD). These efforts are discussed in this paper.

An evaluation methodology was developed based on the concept of fitness-for-use as measured by the construction of architecturally representative applications within a laboratory environment. This method was used to evaluate dozens of commercial software development tools in order to select specific tools as corporate-wide standards.

This work presents the specifics of our software technology evaluation methodology, including our research efforts, tool taxonomy, and evaluation procedures (especially our use of software architecture-style-derived certifying test suites). This paper does not present the specific tools selected through the application of this methodology.

2. SOFTWARE TECHNOLOGY EVALUATION

Many evaluation techniques are known and meet with varying levels of success. Weighted averaging, benchmarking, figures of merit, etc., each have certain advantages and disadvantages (Kontio, 1995). Our approach is instead centered on the concept of demonstrated fitness for use in the environment of choice as measured by the applicability of any given tool to the dominant software architectures found within the target business environment. This approach reflects the "habitat models" suggested by Brown (1996).

This approach stems from viewing evaluation of software from the question: How well does the provided functionality of a product span the needs associated with tasks to be performed using it? Evaluation is highly dependent on the use for which the product is intended and the results are subject to greater ambiguity than evaluations of other classes of products. Many manufacturers of software products will be more than happy to provide metrics for common performance criteria. Other questions are more subtle - does the tool provide the right abstractions, is it easy to use, does it take one hour to do something or ten days. It is these subtle metrics that we intended our evaluation environment to measure and for this we turned to Architecture Styles.

3. SOFTWARE ARCHITECTURE STYLES

A year long study of our software systems identified (at least) four basic architectural styles present in our business applications (Belanger, et. al., 1996). These styles are: transaction, data streaming, real time, and decision support. These styles consistently appeared, in part and in full, in a wide variety of systems including those for Financial, Maintenance, Provisioning, and Asset Management domains. We say, in part, because a majority of our systems are actually hybrids of these different architectural styles.

We eventually derived several certification applications from these styles in order to drive our evaluation process. Our core reasoning being that the development of small scale applications modeled after our target development tasks would prove the suitability of the product under evaluation. This turned out to be true for virtually all the products we evaluated. The entire process of which the architecture styles play a key role is now presented in detail.

4. THE EVALUATION PROCESS

Our approach to evaluating software technology is to appraise technology as "fit-for-use" if we can succeed in developing a sample application which has a reasonable similarity to our production applications. In other words, we use the product under evaluation in an environment modeled after the target development environment. The process can be summarized in the following manner:

1. Survey the available products
2. Classify according to a technical framework
3. Filter the list using screening criteria
4. Construct evaluation criteria templates
5. Use the target tools to build an ***Architecturally Representative*** Application
6. Record findings against the templates
7. Judge the best scores and select the recommended product

4.1 Survey the available products

The overall evaluation process begins with surveying the tool market for candidate products and classifying them according to a technical framework sometimes called a taxonomy. Consider the survey effort first.

Initial research into software tool availability, capabilities, and trends, can be both rewarding and daunting. The goal of tool research is to identify all or most of the tools currently available for the support of a particular stage of the software development process. This research is technical in that one must understand the technological capabilities of each tool. At the same time, this research is market oriented in that one must also understand trends and supplier positioning. Some of the techniques used in this activity include:

- **Literature Reviews:** Books, journals, trade press publications. Key information on technical capabilities, product announcements, corporate changes, tool assessments and recommendations are readily available.
- **Trade Shows and Technical Conferences:** We have found trade shows to be *decreasingly* helpful in identifying technologies of interest. This is due to the generally poor level of technical information available at such venues. Technical conferences on the other hand *remain* helpful in putting the available products into a theoretical or practical context.
- **Direct Mail:** Believe it or not this is an effective means for collecting information once you are on enough mailing lists. (This may not be ecological but it is economical in terms of time; it only takes a few seconds to sort incoming product information.)
- **Automated Topic Searches:** We receive weekly or monthly summaries extracted from current publications on software technologies and trends via email.
- **Web Browsing:** This has become a significant source of information and freeware tools. We maintain a list of vendor web sites and this has often provided up to the minute information on particular products.
- **Vendor Demonstrations:** Slicing through the sales pitch to the technical meat is often difficult but this remains an effective means of collecting detailed product knowledge for selected tools.

- **Evaluation Copies**: A time or event determined interval of hands-on experience, execution, and utilization of the tools is invaluable in understanding actual tool capabilities (this is discussed in detail below).
- **Professional Information Services**: Several organizations are under contract to us providing strategic information on the software industry. This information is often helpful but can also be factually incorrect or misleading. These sources are useful more as sounding boards than anything else.
- **Private Contact Network:** Having a wide network of software professionals to draw upon for knowledge of the industry and technology cannot be overlooked in research efforts. For example, teaching a continuing education course at a local university has brought several new tools to our attention through conversations with students.
- **Experience:** Having been around the development community for a number of years directly impacts your ability to scan and decipher information on tools. Oftentimes "new" tools end up being familiar tools refaced.
- **Project Reference**: Having access to the real life trials of hundreds of development projects we know early on what is needed, what works, and what provides less than advertised.

The output of this research includes summary information on current product availability, industry trends, software standards and standards activities, computing techniques and methods, and development resources both internal and external. The specific products or technologies identified during our research efforts are given an initial classification in the tool and technology taxonomy discussed next.

4.2 Classify according to a technical framework

A Software Development Environment (SDE) can be viewed as an integrated set of tools and processes enabling analysts, designers, programmers, and testers to collaborate on the production of high quality software solutions. Traditional Software Engineering Environment (SEE) frameworks support the concept of creating an SDE by creating a view of the computing infrastructure as a unified and sensible environment with specified functional interrelationships instead of just a random assortment of tools (Brown, 1992).

Unfortunately, SEEs are not well suited to the task of tool classification since they are operational in nature. We required a classification scheme to build our SDE recommendations that could be used to organize toolsets of an eclectic nature resulting from our market research. Existing tool taxonomies (Kara, 1995; Fugetta, 1993; Sharon, 1993) typically focused on particular application domains, limited platforms, or were designed to cover only CASE tools. Since these taxonomies did not meet the needs of our scope (multi-platform, process driven tool standards), we derived our own classification for software tools.

To begin with our classification scheme inherited some structure from our corporate context. Domains typical of most software engineering environments sometimes fall outside of our mission charter. For example, operating systems, databases, and communications protocols are defined by other AT&T teams. Our mission was limited to a constrained view of Application Development technologies.

We decided to base our tool classification on an existing software engineering framework (Utz, 1992) and then modify it as needed (**see Figure 1**). The major categories provided by Utz are re-defined by us below. Each of these major categories are further detailed into sub-categories. Representative sub-categories are shown in **Table 1**. As our market research efforts turn up tools, we categorize them in the taxonomy. Currently we have approximately 1,000 tools in a database organized by these categories. This database allows us to perform ad hoc queries on tool use within AT&T and to quickly produce candidate lists when evaluation efforts are begun.

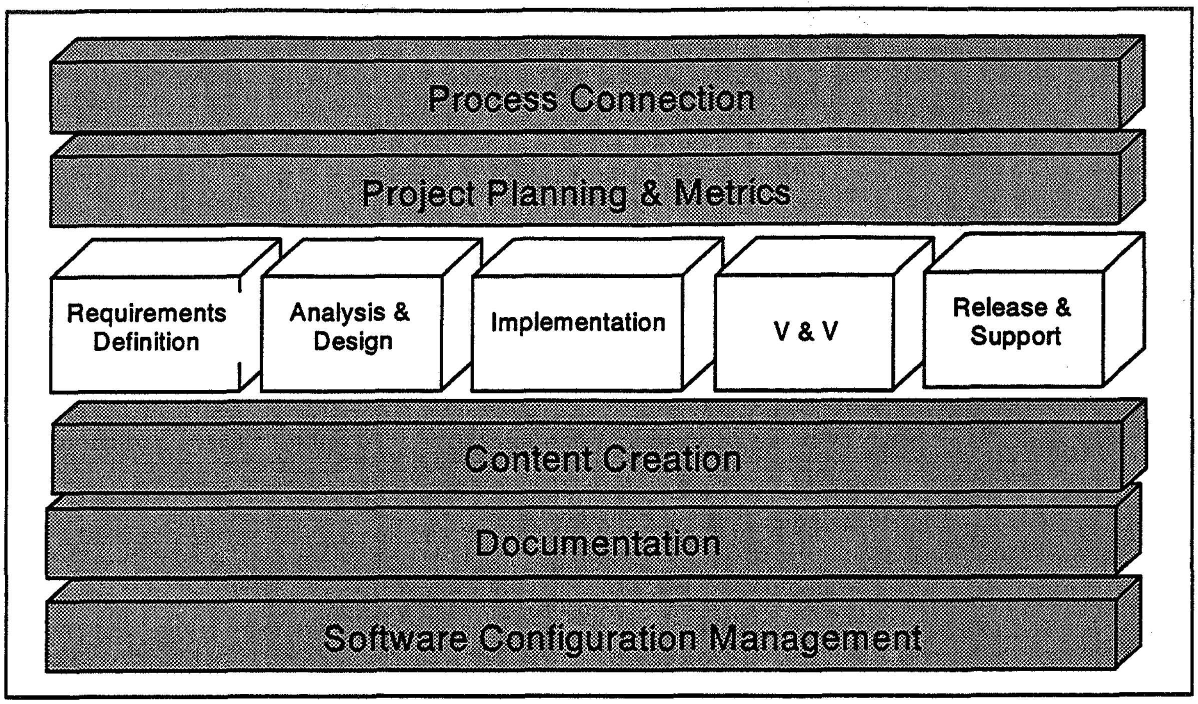


Figure 1: Software Engineering Environment Framework as Tool Taxonomy

4.2.1 The framework categories defined

- **Process Management**: Tools supporting the specification, implementation, and compliance management of development processes.
- **Management & Metrics**: Tools supporting the planning, tracking, and measuring of software development projects.
- **Requirements Definition:** Tools supporting the specification and enumeration of requirements.
- **Analysis & Design:** Tools supporting high level design and modeling of software system solutions following specific formal methodologies and often including code generation and reverse engineering capabilities.
- **Implementation (Code/Debug):** These tools allow both low level code implementation to support the edit-compile-debug cycle of development in 3GLs and visual based programming targeted at rapid application development by use of screen painters/generators with graphical pallets of reusable GUI components with 4GLs.
- **V&V**: Tools providing software verification and validation, quality assurance, and quantification of reliability. These include test case management, test selection, and automated test support.
- **Release & Support:** Tools targeted at supporting enhancements and corrections to existing code as well as browsers, source code analyzers and software distribution.
- **Content Creation**: Tools used for developing Internet materials such as electronically published documents, graphics and multimedia components of Internet sites.
- **Documentation:** Tools supporting creation and distribution of system documentation, specifications, and user information. These tools include documentation storage, retrieval, and distribution.
- **Software Configuration & Manufacturing:** A broad class of tools related to the control of software components and development artifacts including documentation for the purpose of team based programming, versioning, defect tracking, and software manufacturing and distribution.

Process
- Process Definition & Compliance

Project Planning & Metrics
- Project Planning
- Function Points
- General Metrics

Requirements & Definition
- Requirements Trace

Analysis & Design
- Object Oriented Analysis & Design
- Structured or Other Design Methods
- RDBMS Modeling

Implementation
- Languages
- Editors
- Compilers & Debuggers
- IDEs
- GUI/Visual Development
- Cross Platform Development
- Database Development
- Components

Verification & Validation
- Test Management & Design
- Record & Playback
- Stress, Load & Performance
- Coverage

Release & Support
- Distribution
- Reverse Engineering
- Emulation
- Utilities

Content Creation
- Web Document Authoring
- Graphics Authoring
- Multimedia Authoring

Documentation & Workflow
- System Documentation
- Help Authoring
- Workflow

Software Configuration Management
- Source Code Control
- Defect Tracking
- Configuration or Manufacturing
- Integrated SCM

Table 1: Selected Tool Taxonomy Sub-Categories

4.3 Filter the list using screening criteria

With a thousand tools in the taxonomy we have to start trimming the list whenever a particular technology sub-category must be evaluated. Using basic technical requirements many candidate tools can be eliminated. Platform support, negative reviews in the trade press, vendor instability or financial losses by a vendor can all be used to quickly eliminate certain products from the evaluation list. If negative criteria do not work we use positive criteria: is the tool "Editor's Choice" or does our development community already use it as a de facto standard? These types of tools need to be on the evaluation list while others should be dropped.

4.4 Construct evaluation criteria templates

Each of the tool categories in the taxonomy needs specific evaluation criteria to measure the relevant attributes of each tool type in our taxonomy. Towards that end a set of templates must be developed for each type of technology evaluated. These templates resemble the ones found in many trade journals and bench-marking reports. The following must be created or reused:

1. First, one overall template for generic tool and vendor measurement is provided. This generic template covers such items as documentation, support, pricing, and platform availability. A standard set of issues regarding tools such as iconic design, menu features, ergonomics, printing, and so on, is included.
2. Each analyst must then define a specific template which covers the technical aspects of the particular class of tool under investigation, if it does not already exist in our repository of templates. This must be created for each category.

4.5 Use target tools to build Architecturally Representative Applications

Recall that we are interested in demonstrating "fitness-for-use". To do this we now build a representative application with the product(s) selected for evaluated from the taxonomy. Before evaluating any software technology we must first consider what capabilities it has and how to construct a suitable test suite or if our current set of application specifications will need expansion.

4.5.1 Technologies and Their Tasks

Each type of software product dictates certain kinds of tasks that will be the subject of evaluation. For example, word processors might be evaluated in terms of developing on-line (in program) documentation, help files, man pages, hard copy user manuals, and HTML documents. On the other hand, one would not evaluate a compiler in terms of its support of those same tasks. In some cases, products span more than one functional category. For example a C++ IDE might provide a visual programming environment, a class system, and a general purpose compiler. Since each of these is a separate endeavor, an evaluation of a C++ IDE will concentrate, independently, on the visual programming environment, class system completeness, and compiler performance. These are individual and discrete evaluations. Each will need specific resources to carry out the evaluation.

4.5.2 Software Resources for Evaluation

The software resources required to complete the data collection demanded by the evaluation template fall into three categories: 1) the software under evaluation; 2) supporting software (i.e., the operating system); and 3) software in the form of test cases (e.g., a sample design to implement). As we have shown, common architectures run through most AT&T applications. Our concept was to derive the required test cases from these architecture types or patterns.

Software patterns (Gamma, 1995; Coplien, 1995) formalize some of the concepts on recurring underlying software construction themes. We devised evaluation test cases to demonstrate that any tool recommended supported AT&T's specific computing problem domains. Thus we developed and specified a set of representative applications modeled after architectural styles or patterns observed in the field, to serve as certifying test suites for any tool slated for review (see **Table 2**).

Arch/UI Style vs. Sample Apps	ARCH STYLE				GUI STYLE				
	OLTP	Data Stream	Decision Support	WWW	Forms	Active Graphic	Alert Panel	Map Based	Hypertext Browser
Contact	X		X		X				
COD		X					X		
GEM		X				X			
NetAnalyst			X			X		X	
ToolBase	X		X	X	X				X

Table 2: Representative Applications and their Architecture Styles

The representative applications and their relationship to the generic architecture styles of **Table 2** are briefly described below:

- **Contact Data Base:** The Contact Data Base is a very simple system for managing contacts on a project-by-project basis. Contacts are managed at the level of tracking individuals associated with a project, individual meetings, and tracking tools employed on the project. This application demonstrates a forms based interface for data entry and reporting.

- **Co-Operative Document System (CODS):** The Co-Operative Document System allows multiple people to work on the same document. The basic capability of checking a document into and out of a document control system is augmented with a message broadcasting feature alerting users of a subscribed document's state. This represents a client server system with data streaming and on-line transaction architectural components.
- **Graphic Enterprise Modeler (GEM):** The graphic organization display provides the ability to model graphically the structure of a corporate organization. It visually illustrates relationships among people, projects, and teams. To find answers to specific questions regarding an organization, the user follows semantically meaningful links and uses active graphics controls. This application demonstrates the user interaction style of the active graphics variety.
- **NetAnalyst**: This application is a map based data visualization tool. It takes a set of real telecommunications data (the 1994 L.A. earthquake phone traffic) and plots it geographically. This is a common type of application profiling decision support and mapping.
- **ToolBase**: This is an Intranet based front end to a product tracking database. This application provides for the evaluation of many types of Internet technologies and the extent to which they can support the architectural styles of OLTP and decision support on the Intranet.

Returning to our evaluation process, an appropriate application is selected to test the tool class and development against a set of specifications describing the sample application is begun. Often, the specifications need modification or additional software design efforts need to be conducted to fully stress the products under evaluation (e.g., our Internet application did not test multimedia features as initially designed). Inferior products fail during implementation of the specifications and quickly drop out.

4.6 Record findings against the templates

Throughout the work of building the sample application, feature performance data must be captured on the custom template constructed for this technical category. This includes objective and subjective measures. Subjective data includes how intuitive the product was or how friendly the help desk was when called. Objective data includes if the promised features worked and if you could accomplish the task of building the sample application.

Weighted Scoring Method (WSM) is normally used to provide a simple rating mechanism for each product under evaluation. In this method each item in the criteria matrix is assigned a score or weight score. Usually a score of 1 to 5 is given to the product for each criterion. Then an overall score can be derived using the formula below (Konito, 1996):

$$\text{Score}_a = \sum_{j=1}^{n} (\text{ weight}_j * \text{score}_{aj})$$

4.7 Judge the best scores and select the recommended product

The final step is recommending a product. Out of the short list all products are evaluated. Using the sample application as a test suite the superior product normally emerges. With a WSM technique there is very small opportunity for any ties. The analyst must, however, still exercise their best judgment in selecting a product for recommendation.

5. EVALUATION PROCESS RESULTS

Within a laboratory environment we developed these representative applications repeatedly using different software technologies. We also carried out other tasks in support of this simulated development work, such as configuration management, using still more products under evaluation. This approach provided clear evidence of the suitability of one product over another and was much easier to derive than by only looking at a feature capability matrix. We had a high degree of confidence that the product would work on a real development project using this method.

Dozens of tools have been evaluated using this method and still others are currently under examination. From this work many standard products have been chosen that are now part of AT&T's overall body of internal technical standards. Through controlled introduction using pilot projects and consultative jump-starts many of these products have also proven to be successful on large-scale software projects. Recently this technique was also used successfully to evaluate over 30 software products used in Internet based development projects.

6. PORTING THE PROCESS

Deployment of this technique to a different environment requires minimal modifications. We have reused this process from the evaluation of Windows based tools to the evaluation of Internet based tools seamlessly. To transfer this process to a different development base or user community we recommend making the following changes:

1. The tool taxonomy must be recalibrated to fit your environment and goals. Our taxonomy does not address databases, office automation, or operating systems. You need to add the appropriate technologies to fit you computing framework.
2. Your architectural styles may vary from ours. We develop very few "hard" realtime systems or embedded systems of any kind since our spin-off of Lucent Technologies. There may be other significant architectural styles you will need to identify.
3. After adjusting the framework and architectural styles you now need to document your screening criteria and create your detailed evaluation criteria templates. A good template typically requires a couple of days for an analyst to create. They are reusable and typically only one is necessary per technical category.
4. *Execute.* This is the crucial step where the watch-word is "emulation". That is, emulation of your actual development process and tasks.

We are confident that by following these simple steps the process we have been using for the last two years can be re-deployed in any software development technology evaluation laboratory.

7. CONCLUSIONS

Using applications derived from clearly relevant architectures keeps the evaluation process honest. Analysts with development backgrounds typically feel more comfortable building an application than acting as a software critic. Simulating the development tasks in this way does not solve all the problems with technology evaluation. Politics and compromise are inescapable factors when making decisions that will commit a corporation to spending or not spending large sums with any given vendor. Also, some variability remains in the scoring technique. Each analyst tends to have peculiar habits in working through a 200 item feature matrix. One may score "high" or "low" while another may include "medium". Nevertheless, we feel confident that architecture styles add a healthy modicum of extra validity to the otherwise typical process we have described.

8. ACKNOWLEDGEMENTS

Naturally work of this type cannot be accomplished without the cooperation and support of dozens of people. We are indebted to several senior managers for their enthusiastic top-down support of this work including Judy Page and Dick Machol of NSD, and Rod Mack and Illene Hochman of BMD. Our immediate support managers, Barbara Beech and Moses Ling, were especially helpful in guiding this effort. Finally, we would like to thank the dozens of people who provided information, feedback, and carried out the evaluations across AT&T and NSD. In particular we appreciate the work of the NSD Software Development Environment Team and the AT&T Foundation Architecture Software Development Tools Team. This paper reports on the combined work of each of the individual team members.

9. REFERENCES

1. Belanger, D., et. al., *"Architecture Styles and Services: An Experiment on SOP-P"*, **AT&T Technical Journal**, Jan/Feb, 1996, pp54-63.

2. Brown, et. al., **Software Engineering Environments: Automated Support for Software Engineering**, McGraw-Hill, 1992.

3. Brown, et. al., *"A Framework for Evaluating Software Technology"*, **IEEE Software**, September 1996, pp39-49.

4. Coplien, J., & Schmidt, D., eds., **Pattern Languages of Program Design**, Addison-Wesley, 1995.

5. Fuggetta, A., *"A Classification of CASE Technology"*, **Computer**, Dec. 1993.

6. Gamma, et al, **Design Patterns: Elements of Reusable Design**, Addison-Wesley, 1995.

7. Kara, D., *"Client/Server Development Toolsets: A Framework for Evaluation and Understanding"*, **Application Development Expo**, New York, NY, April 4, 1995.

8. Konti, J. and Tesoriero, R, *"A COTS Selection Method and Experiences in its Use"*, **Proceedings of 20th NASA Software Engineering Workshop**, Greenbelt, MD, November, 1995.

9. Konti, J., *"A Case Study in Applying a Systematic Method for COTS Selection"*, **Proceedings of 18th International Conference on Software Engineering**, Berlin, Germany, March 25-26, 1996.

10. Sharon, D., *"A Reverse and Re-Engineering Tool Classification Scheme"*, **IEEE Software Eng. Tech. Committee Newsletter**, Jan. 1993.

11. Utz, W., **Software Technology Transitions: Making the Transition to Software Engineering**, Prentice Hall, Englewood Cliffs, NJ, 1992.

21st NASA SEL Software Engineering Workshop

SOFTWARE DEVELOPMENT TECHNOLOGY EVALUATION: PROVING FITNESS-FOR-USE WITH ARCHITECTURAL STYLES

December 4, 1996
Greenbelt, MD

James Cusick
AT&T
Bridgewater, NJ
James.Cusick@att.com

William Tepfenhart
AT&T
Middletown, NJ
William.Tepfenhart@att.com

Copyright © 1996 AT&T

TODAY'S TALK

- Concepts for Architecturally Driven Evaluation
- The Evaluation Process
- Examples and Results
- Conclusions

Copyright © 1996 AT&T

SOFTWARE TOOLS AND STANDARDS AT AT&T

- Gartner Group estimates 40,000 software tools on the market
- AT&T Has Hundreds of Projects Ongoing at any One Time
- Training, Integration, Portability, Quality Drive Standards

*HOW WOULD <u>**YOU**</u> CHOOSE*
A FEW DOZEN TOOLS
FOR CORPORATE WIDE
DEVELOPMENT NEEDS?

Copyright © 1996 AT&T

EVALUATION APPROACHES REVIEWED

<u>Informational</u>

✓ • Case Study Reviews
✓ • Questionnaires
• RFI
• Vendor Demos
✓ • Published Reviews

<u>Experimental</u>

✓ • Weighted Averaging
• Benchmarking
• Figures Of Merit
✓ • Sample Applications
• Pilot Projects

✓ = *Techniques we Favored*

Copyright © 1996 AT&T

ARCHITECTURE STYLES INTRODUCED

What is an Architecture Style?

A set of operational characteristics common to a family of a software architecture and sufficient to identify that family.

AT&T study yields four dominant styles :

* Transaction
* Data Streaming
* Real Time
* Decision Support

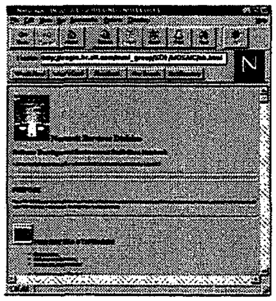
* <u>*Most systems are Architectural Hybrids*</u>

Copyright © 1996 AT&T

MORE ON ARCHITECTURE STYLES

USER INTERFACE STYLES

- Forms
- Documents
- Active Graphics
- Alert Panels (ie, mail program)
- Maps
- Hypertext



Copyright © 1996 AT&T

EVALUATION PROCESS SUMMARIZED

1) Survey the available products

2) Classify according to a technical framework

3) Filter the list using screening criteria

4) Construct evaluation criteria templates

⇨ 5) ***Use tools to build Architecturally Representative Applications***

6) Record findings against the templates

7) Judge the best scores and select the recommended product

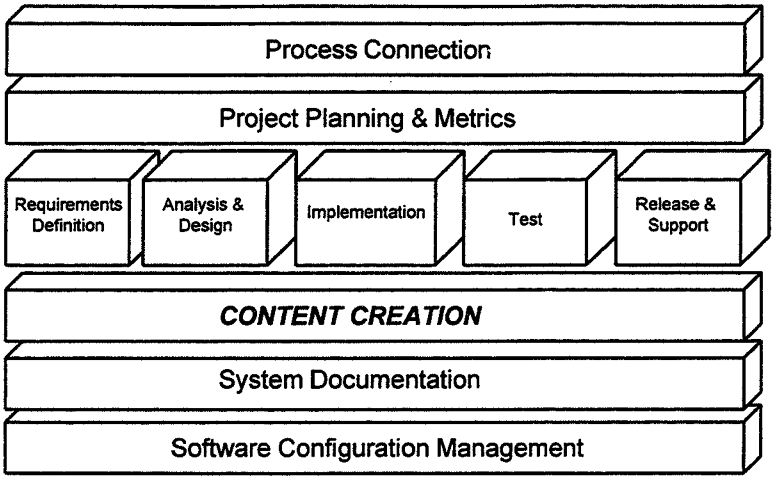
Copyright © 1996 AT&T

SURVEY TECHNIQUES

- Literature Reviews
- Trade Shows and Technical Conferences
- Direct Mail
- Automated Topic Searches
- Web Browsing
- Vendor Demonstrations
- Evaluation Copies
- Private Contact Network
- Experience
- Project Reference

Copyright © 1996 AT&T

AN OVERALL TOOL FRAMEWORK


Process Connection

Project Planning & Metrics

Requirements Definition

Analysis & Design

Implementation

Test

Release & Support

CONTENT CREATION

System Documentation

Software Configuration Management

Utz, 1992

Copyright © 1996 AT&T

SELECTED TAXONOMY SUBCATEGORIES

PROCESS/PLANNING/METRICS/REQ.
- Process Definition & Compliance
- Project Planning
- Function Points
- General Metrics
- Requirements Trace

ANALYSIS & DESIGN
- Object Oriented Analysis & Design
- Structured or Other Design Methods
- RDBMS Modeling

IMPLEMENTATION
- Languages
- Editors
- Compilers & Debuggers
- IDEs
- GUI/Visual Development
- Cross Platform Development
- Database Development
- Components

V&V
- Test Management
- Test Design & Generation
- Record & Playback
- Stress, Load, & Performance
- Coverage

RELEASE & SUPPORT
- Distribution
- Reverse Engineering
- Emulation & Utilities

DOCUMENTATION & WORKFLOW
- System Documentation
- Help Authoring
- Web Authoring
- Workflow

SCM
- Source Code Control
- Defect Tracking
- Configuration and Manufacturing

Copyright © 1996 AT&T

FILTERING CANDIDATES & BUILDING CRITERIA

FILTERS

- Platform
- Language
- Corporate Requirements

CRITERIA

- Product Specs
- Journal Reviews
- Existing Templates

Copyright © 1996 AT&T

SELECTING A CERTIFICATION APPLICATION

Arch Style vs. Sample Apps	ARCH			STYLE	
	OLTP	Data Stream	Decision Support	WWW	Real Time .
Contact	X		X		▨
COD		X			▨
GEM		X			▨
NetAnalyst			X		▨
ToolBase	X		X	X	▨

UI Style vs. Sample Apps	GUI			STYLE	
	Forms	Active Graphic	Alert Panel	Map Based	Hypertext Browser
Contact	X				
COD			X		
GEM		X			
NetAnalyst		X		X	
ToolBase	X				X

Copyright © 1996 AT&T

RESULTS: SCORING & RECOMMENDING

- Recursive Development Efforts Yield Feature Scores
- Simple Weighted Average Applied

$$\text{Score}_a = \sum_{j=1}^{n} (\text{weight}_j * \text{score}_{aj})$$

- Scores + Objective Side-by-Side Performance on Sample Application Determine Recommendation

Copyright © 1996 AT&T

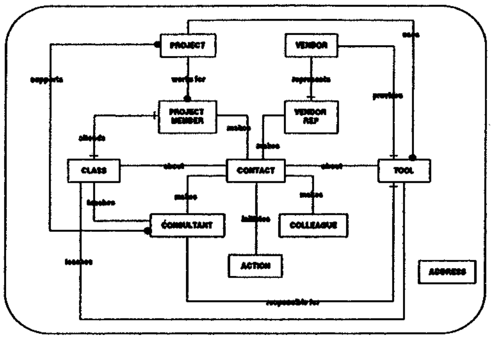
AN EXAMPLE: RDBMS MODELING

- Needed RDMBS Reverse Engineering & Modeling
- Selected CONTACT Application
  - Reuse GUI Forms and DB created for earlier eval
  - Good Reverse Engineering candidate
  - Modify Schema and Rehost on new RDBMS
- Many integration, support, and administrative problems

Copyright © 1996 AT&T

REQUIREMENTS & ARCH STYLE

Table of Contents



CONTACT Information Model

CONTACT Certifies:
OLTP + DSS + Forms

Copyright © 1996 AT&T

IMPLEMENTATION RESULTS

Where is stability and multiple database support?

Analyst: I choose Access ODBC. If I use a name such as "customer name" (note the space) as a field name for an element in a record, I'll get the message "invalid field name" while I generate schema. However, I can create a table with a field name "customer name" directly in Access. Is this a problem in TOOL-ABC?

Vendor: I need to try MS Access Jet ODBC instead of using Access 2.0.

Analyst: I choose Watcom 4.0. In database engine, after a schema generating, I want to change some of my records. When I choose a record and click "Edit", TOOL-ABC exits automatically and goes to the DOS prompt. After restarting Windows project is now "Exclusively locked". Project cannot be deleted or renamed.

Vendor: In order to recover my project, I was instructed to go to the project directory from Windows File Manager, then delete some files and copy other files, etc. No reason given for the problem.

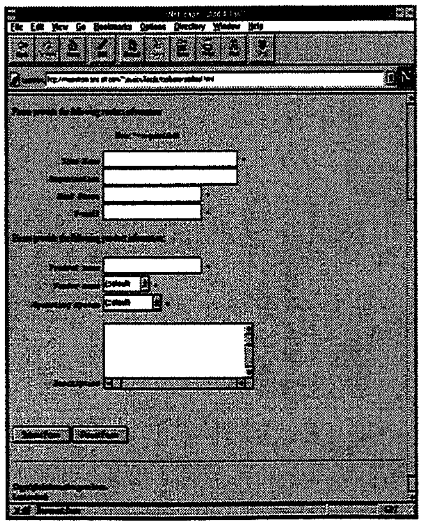
Copyright © 1996 AT&T

EXAMPLE 2: HTML AUTHORING FOR DEVELOPMENT

- Needed Update for HTML Authoring Recommendations
- Created New Application: ToolBase
  - Existing forms based OLTP/DSS Application
  - Redesign for Hypertext Browser
  - Implement as interactive WWW DB app
- Realized Need For Additional Modifications:
  - Originally built as simple UI
  - Re-fit with extensive images to test graphics toolkits

Copyright © 1996 AT&T

REQUIREMENTS & ARCH STYLE



Contents

Database schema
Selection forms
Report layouts
Usage descriptions
Table definitions
HTML prototype

ToolBase Certifies:
OLTP + DSS + WWW
Hypertext Interface

Copyright © 1996 AT&T

IMPLEMENTATION RESULTS

STOP

No Support for Database Connectivity

Poor Selection of GUI Widgets

Limited Visual Alignment Capabilities wrt Req.

Generally Poor support of Native HTML Editing

Copyright © 1996 AT&T

ASSESSMENT TOTALS PER CATEGORY

TOOL TYPE	COMPLETED EVALS
• GUI Development	3
• RDBMS Modeling	4
• GIS	2
• HTML Authoring	6
• Java IDEs	4
• Web Database Tools	2
• Software Testing Tools	4
• SCM	4

Copyright © 1996 AT&T

BENEFITS & CHALLENGES

☺ Ties Evaluations to Actual Development Tasks

Produces Representative Apps to Daisy-Chain Evals

Supports Difficult Decision Making Task with Objective Data

Creates Excellent Demonstrations for Consulting

☹ Does Not Provide Escape from Politics

Process Requires Some Education for Each Participant

Some Variability and Subjectivity Remains
(especially in applying consistently and scoring techniques)

Copyright © 1996 AT&T

CONCLUSIONS

Advising ourselves on starting over:

- Establish dedicated lab space
- Secure superb technical support
- Rotate talent
- Assure top-down management support
- Expand internal communication efforts
- Invite more vendor "bake-offs" (let them build it!)

- *Stay the course on architecturally relevant samples*

Copyright © 1996 AT&T

Systematic Process Improvement in a Multi-site Software Development Project

H. Hientz, G. Smith, A. Gustavsson,
P. Isacsson and C. Mattsson
{hh, gs, agu, pi, cma}@q-labs.com

**Q-Labs Software Engineering GmbH,
Germany**

55-61
415648
360697
p14

Abstract This paper reports on the application of the PERFECT[1] Improvement Approach and specifically goal-oriented measurement via GQM[2] in a large multi-site software development project. Successful and persistent implementation of an Experience Factory and the GQM approach in a large multi-site project organization is a challenging task and needs to be based on a sound and operational methodological support to face all the practical problems and resistance which occur in the course of a software process improvement programme. In the paper we present both measures and experiences of applying goal oriented measurement as well as experiences from introducing systematic process improvement based on measurement.

Keywords: Systematic process improvement, software measurement, Goal Question Metric paradigm, Experience Factory approach

1. Introduction

When introducing persistent process improvement in an organization there is a need for having an underlying framework for what activities that need to be carried out in order to get lasting results. New results e.g. the Experience Factory [6] and GQM [2] points to the need of introducing:

- explicit modelling of products, processes and quality aspects in order to understand the building blocks in software development and to be able to tailor them for specific needs, measure their adherence within different projects and to improve them across projects.
- comprehensive reuse of models, knowledge and experience in order to choose appropriate models for new projects and to compare actual project data with baselines.
- measurement integrated with the software development in order to define quality goals, understand differences between projects and to control whether quality targets have been met

In this paper we report on the experiences in establishing such a process improvement program using the PERFECT project [1] results as a methodology basis. PERFECT should be viewed as one possible instantiation of the Experience Factory concept and provides a more detailed description of how to implement the process improvement framework. The organization described in this paper already promoted explicit modelling of products and processes. The next obvious implementation step was to integrate goal oriented measurement to create better understanding of the current baselines and thus in the subsequent projects better facilitate the future reuse of experiences and achieve improvements across projects and sites. This document describes the results of introducing goal oriented measurement and the first implementation steps in order to set up an experience factory.

The paper is organized as follows. In chapter 2 we give the overview of the application project, its characteristics and organization. Chapter three introduces the process improvement framework that was used. Chapter 4 focuses on how goal oriented measurement was applied using the GQM approach [2]. The used method is described in detail together with examples from the collected measures and the analysis. Finally, in chapter 5 the conclusions are presented.

2. The Multi-site Improvement Project

The target for the process improvement was a software development project of 350,000 m.hrs of effort over one year for the development of a new release of a product in

[1] Process Enhancement for Reduction of software deFECTs.

[2] Goal/Question/Metric

Ericsson's GSM mobile telephony range. It was a collaborative development involving five separate design centres.

It was the aim to carry out process improvement in a systematic way rather than the ad-hoc approaches that usually characterize process improvement programmes. This means that:

- a systematic model of process improvement was used to provide a framework for the programme,
- the improvements were run within a separate improvement project with its own budget, plans, organization and reporting structure. This project ran 'in parallel' with the target software development project.

What is meant by a "systematic approach to process improvement" is mainly the fact that the programme should be based on established models. In this project, there were several models underpinning the project:

- a process improvement project structure
- a process improvement organization
- the 'PERFECT' Model of Process Improvement [1]
- the GQM Approach for goal-oriented measurement [2]
- an approach to technology transfer

The main structure of the project is illustrated in figure 1, which shows:

- the gathering of experiences from the previous project during the pre-execution phase
- the use of the methodology framework from the 'PERFECT' project, and the feedback of experiences
- the methodology development activities providing improved methods and processes to the technology transfer part
- the interaction with the target project at the five sites through the technology transfer activities
- feedback from the target project
- the analysis activity after target project termination

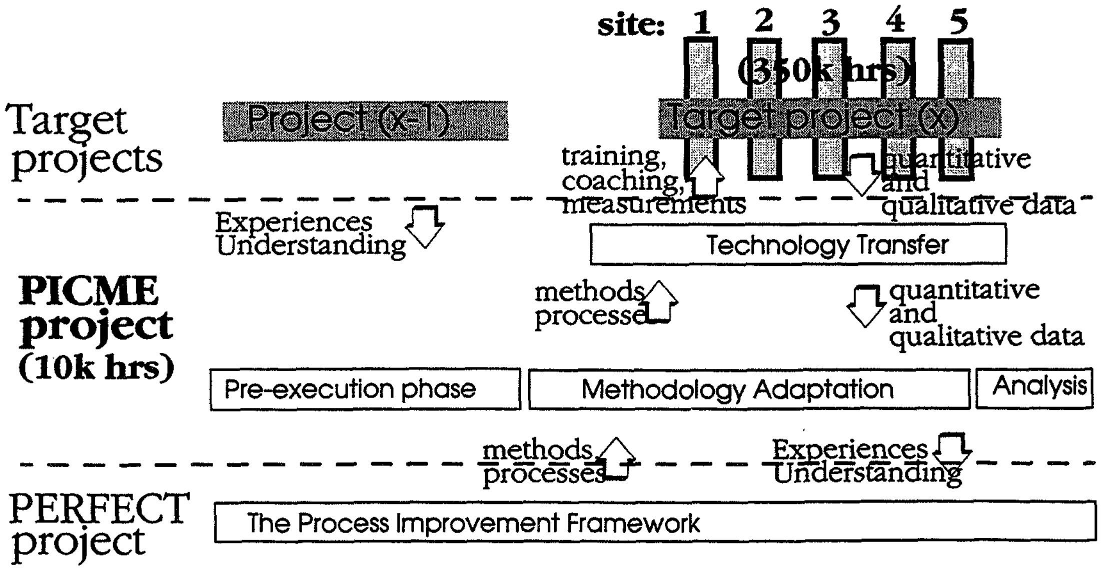
It was necessary to set up an organization to carry through the improvements. To ensure that the process improvement programme maintained close contact with the design teams, process improvement teams (P.I.Ts) were set up in each site. They consisted of project members from that site and their role was to ensure a good two-way flow of information, ideas, and feedback between the process improvement programme and the design teams. The multi-site organization of the project consisted of:

- A multi-site 'Process Improvement Coordination Team' (PICT)
- Process Improvement Teams (PITs) in each site
- Process improvement consultants (Q-Labs)

The Process Improvement Coordination Team (P.I.C.T) was, as the name suggests, intended to coordinate and harmonize the activities across all sites in the project.

This organization was deliberately 'bottom-up', i.e. the driving force behind the programme was intended to be the site PITs to ensure that the improvement proposals accurately reflected the real needs of the users.

Figure 1: The Multi-site project organization



3. The PERFECT Process Improvement Framework

The process improvement approach described in this paper is the result from the European ESPRIT project PERFECT [1], especially the organizational structuring of the improvement project and the goal-oriented measurement parts. The PERFECT project had when it started in September 1993 the goal to package for European industry methods and models for establishing measurement-based initiatives aimed at evolutionary improvement of software development processes relative to company-specific goals.

The PERFECT Improvement approach is based on the technologies developed by Basili et al., the Quality Improvement Paradigm (QIP) on the methodological aspect, the Experience Factory (EF) for the organizational aspect and the Goal-Question-Metric (GQM) method for the goal oriented measurement activities, see for instance [2], [5], [6]. These concepts have by the PERFECT project been detailed and enhanced with activities and packaged for use within the trial-applications of which PICME was one such. At the closure of the PERFECT project all developed methodologies were packaged in booklets as deliverables.

The PERFECT Improvement Approach Experience Factory Model (PEF)

PEF is based on the existing material from SEL as well as not documented experiences. In addition to this we have used the industrial experience from european software industry to adapt and add on necessary areas. The usage within the PICME project gave together with the other applications many useful comments for updating and evolving the PEF when it comes to roles, responsibilities and activities.

In the PEF model the EF (Experience Factory) is one part out of three, see Figure 2. The other parts are: the software development projects that is execution as case studies, supported by the EF; and the sponsoring organization in which the projects as well as the EF resides. All three parts are equally important in establishing an effective improvement initiative.

Experience Packages

The PEF model is based on a focused and simplified model of an Experience Package. It includes three parts: process model, process control model (quality model), and process experience. The first part is what is traditionally handled by training and experts; the second is handling the GQM and measurement parts; while the third is focusing on the actual data, the conclusions and new hypothesis that can be drawn based on the process model, the measurements and the project characteristics.

The EF in PEF

The EF in the PEF model consists of three focus parts, as can be seen in figure 2. One part handles the issues of the overall improvement work (the Strategic Improvement management); one handles the specific issues with each separate Software development project that are supported by the EF (the Project Support); and one handles all specific project results that should be analysed and then generalized/synthesized for the whole organization (the Experience Package Engineering).

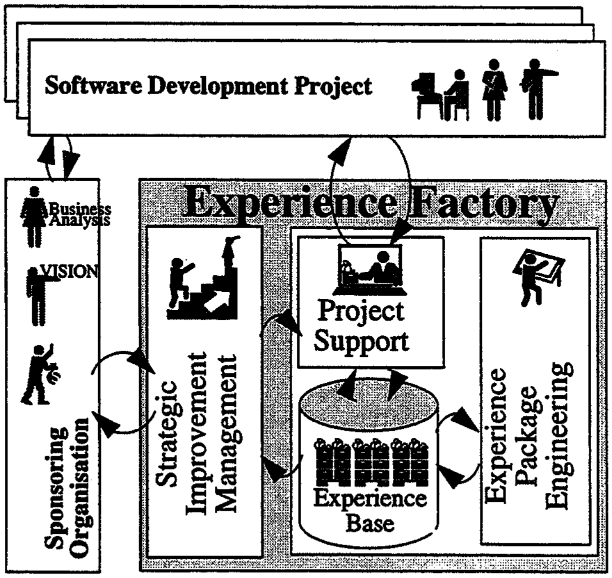
Comparing the EF in the PEF with the NASA/SEL EF

In relation to the NASA/SEL EF there has been put more emphasis on placing the EF into a context within an organization. Especially the modular approach which emphasizes the importance and clarifies the tasks of the different areas both outside and inside the EF.

From the outside and in following could be noticed:

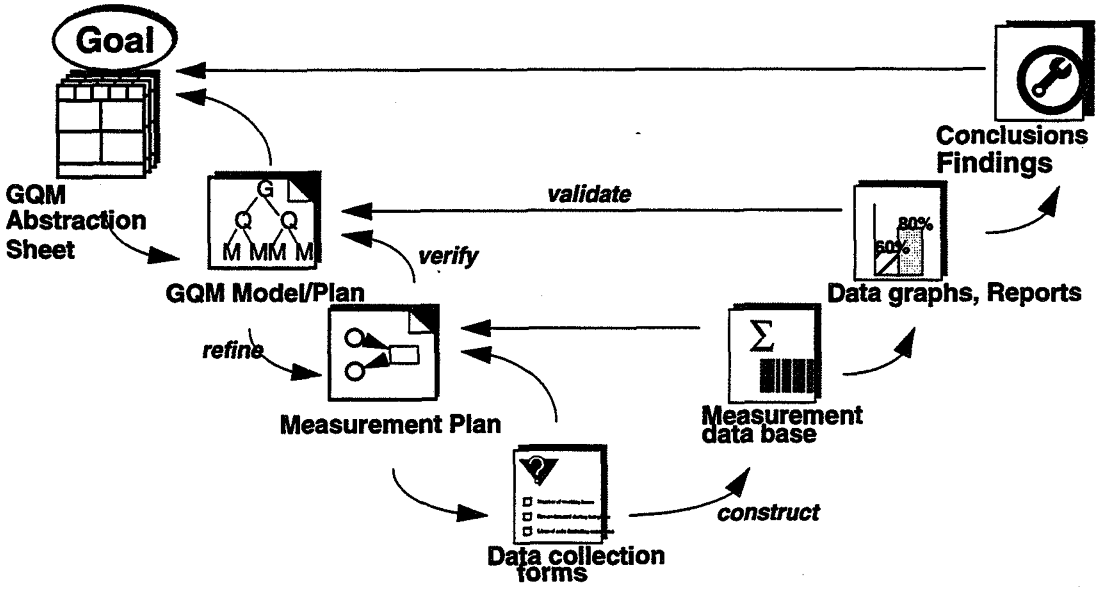
- The roles in the sponsoring organization that are necessary to establish an EF and the improvement initiative have been made explicit. The connection to the business goals and market situation, the internal organizational development and the short term economical interest of the organization are described.

Figure 2: The PERFECT Experience Factory, PEF



- In the projects it is suggested that the project itself should take responsibility for the measurement collection and validation. Measurements must be an active part of managing each project.
- For the EF it is suggested that the overall issues of running the EF also must be addressed more systematically and goal oriented. That includes support for systematically selecting new technologies to experiment with and introduce, the actual running of an improvement programme in a goal driven manner and the handling of change on personnel and organizational level.
- The need for active support of each software project is also highlighted. This is one, often neglected, success factor in process improvement. It is highlighted the need for different kind of support, i.e.: process training and coaching; setting up efficient goal oriented measurement programmes for the project; and support in reusing (identifying, understanding and applying) the experiences (conclusions and hypothesis) from previous projects.

Figure 4: GQM V-Model



- The third part of the EF in the PEF model is the one with direct similarities to the NASA/SEL EF. The distinction here is the structure of the activities, i.e., following the basic structure of the Experience Package: Process Model, Process Control Model and Process Experience.

Experiences from Experience Capturing from Perfect Application projects

Since the PEF model evolved to its current shape from project feedback late in the PERFECT project the applications did not have enough time for a full implementation. When mapping organisational entities and evaluating the activities in the project it is reassuring that activities in the PEF model are either already performed or there has been a need for introducing in the analysed application. Especially promising was also comments like "the PEF model would have helped us organize the improvement initiative better" from one application provider.

Identifying instances of the PEF within the PICME project the PICT could be viewed as the strategic improvement management force and the PITs as the project support functions of the PEF.

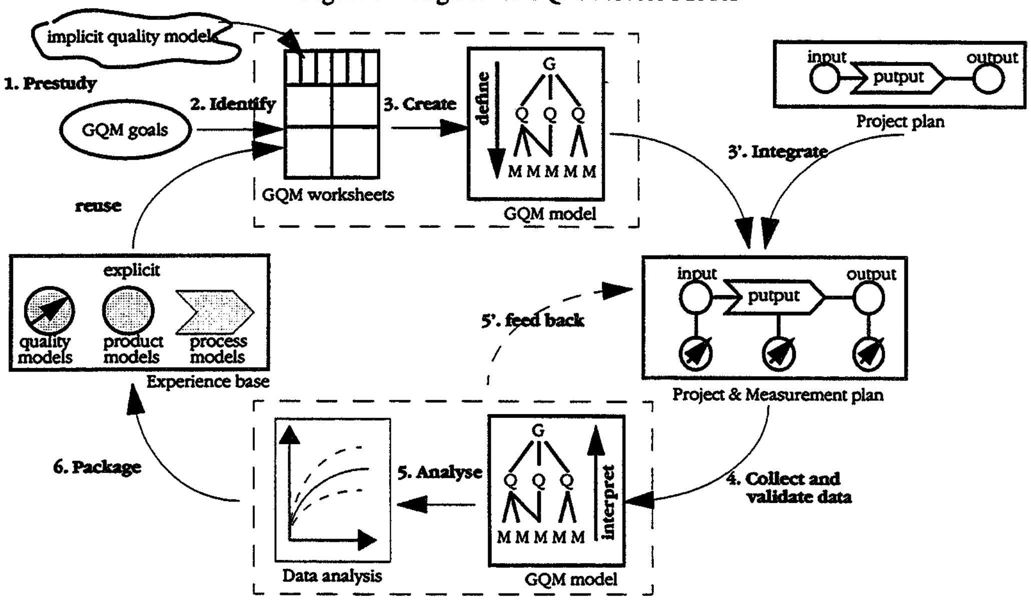
In the PICME-project the PERFECT Improvement Approach, so far it had evolved, was used partly by applying the steps of the QIP and extensive usage of the GQM approach.

4. Goal-oriented Measurement with GQM

The Reference Model

Conducting a measurement programme in a large multi-site organization has to be done following an explicit measurement process, using well defined measurement artifacts and involving a diversity of project staff from management to software development engineers. The GQM V-Model (Figure 4), as defined in and for this project, provides a reference model to illustrate, communicate and guide the measurement programme. It provides the ability to explain and trace the measurement approach followed (ad-hoc/bottom-up versus goal-oriented/top-down), the involved roles (from viewpoint of the analysis task to the data provider), required artifacts (analysis goal to data collection sheets), and key activities such as refinement, verification and validation steps.

Figure 3: High-level GQM Process Model



The Measurement Process

Having the GQM V-Model in place a high level measurement process is used to enact the measurement programme. The underlying measurement process (figure 3) consists basically of six steps: 1) characterization of environment, 2) set GQM measurement goals, 3) develop GQM models and produce measurement plan considering the reuse of existing experiences and models, 4) collect and validate data, 5) analyse data and provide project feedback, 6) package experiences for future reuse. The measurement process varies depending on the purpose of the analysis goal, e.g., assessing a delivered software product versus evaluating a software development process, reusing GQM measurement goals and experiences versus executing a measurement programme from scratch.

The process steps are not followed in a waterfall like way, iterations should be considered, i.e., completion criteria must be defined and checked. A more formal description and experienced details could be found in [1], [9] provides lessons to be learned regarding measurement-based process improvement.

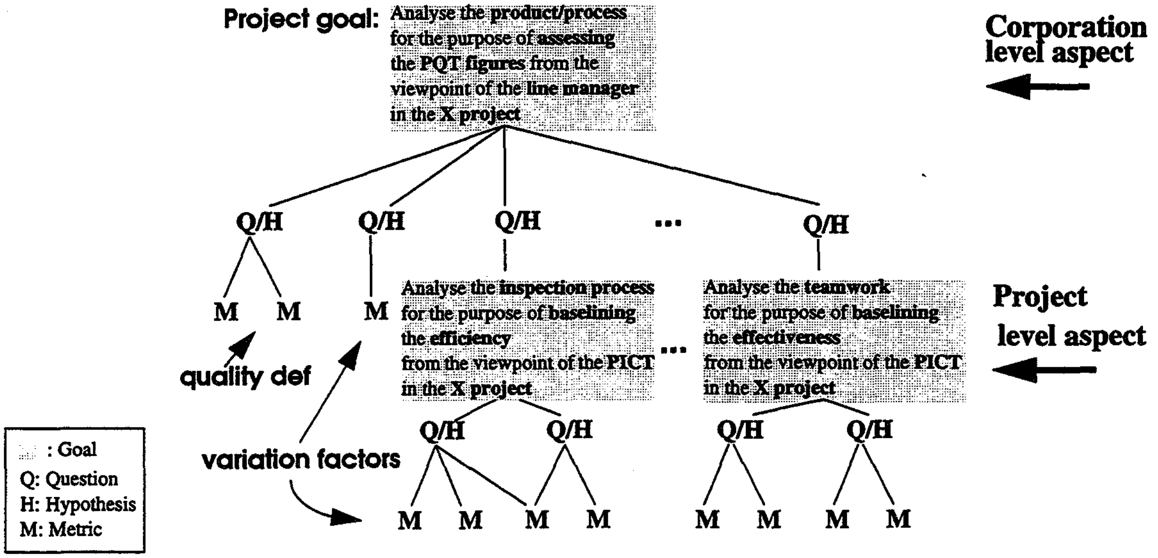
The Measurement Goals

The GQM analysis goals were prioritized according to the improvement goals of the organization, e.g., reduce time to market by 20%, and the catalog of improvement proposals targeted by the improvement programme. The GQM-based measurement goals were integrated into the existing corporate wide measurement programme of assessing the projects performance in terms of Productivity, Quality, and Leadtime. The GQM goals were targeted on Inspection Efficiency, Teamwork Effectiveness, Work allocation Fitness, Stability of Requirements, and Applied Design Process Performance. All these goals were analysed in detail. But, due to the limited scope of this paper and confidentiality reasons only the Teamwork Effectiveness could be reported throughout this paper. The relation between corporate and project concerns was captured in a GQM 'goal-tree' which is illustrated in Figure 5. The underlying software development process (here: prescriptive, waterfall like) must be respected which has a major impact on all the facets of the GQM goal, i.e., object of study, purpose, quality focus, viewpoint, and context. Also the scope of the measurement goals should be constrained based on the resources dedicated to the measurement programme and the organization's maturity. Maturity is defined in terms of stability of the processes in place and the ability to adhere to them.

responsible roles for providing the data. Tools for data collection were either based directly on existing ones, e.g., time reporting system, paper/email-based questionnaires or enhanced existing tools, e.g., inspection record collection tool.

A simple spreadsheet application is sufficient to process all the collected data and aggregate them to the level of data analysis charts. Tool support should respect the principle of GQM, i.e., goal orientation. Currently web technology is being investigated, as part of the 'engine room' concept, which increases transparency and improves the access to FAQ's, glossaries, instructions, etc.

Figure 5: GQM goal-tree


Project goal:
Analyse the product/process for the purpose of assessing the PQT figures from the viewpoint of the line manager in the X project
Corporation level aspect
Q/H
Q/H
Q/H
...
Q/H
M
M
M
Analyse the inspection process for the purpose of baselining the efficiency from the viewpoint of the PICT in the X project
...
Analyse the teamwork for the purpose of baselining the effectiveness from the viewpoint of the PICT in the X project
Project level aspect
quality def
variation factors
Q/H
Q/H
Q/H
Q/H
M M M M
M M M M
▨ : Goal
Q: Question
H: Hypothesis
M: Metric

The GQM Models

GQM abstraction sheets are useful for refining the GQM goals during interview sessions held with viewpoint representatives, affected project staff, and line representatives. The abstraction sheet for baselining the Teamwork Effectiveness is depicted in Figure 6. The derivation of the variation factors (VF) is guided by categories of factors which are considered to have a main impact on the object of study ([2]), e.g., domain conformance as VF 1 and VF 2, process conformance as VF 3 to VF 7. Likewise, the quality focus is defined based on the knowledge of the target environment which is based on the viewpoint's experience.

The Data Collection

Data collection is triggered by periodic activities, e.g., weekly time reporting, process states, e.g., begin/end of phases or entry/exit criteria, and artifact state transitions, e.g., inspected documents. The triggers determine the

Data Analysis

Data analysis was done without involving sophisticated statistical support. Nevertheless, validation of the variation hypothesis (figure 7) and a comparison of the actual data with the baseline hypothesis (figure 8) were performed in regular feedback sessions.

To a limited extent the Rough Set approach was applied ([4], chosen among other approaches, e.g., [7]) to analyse and package the measurement results. The Rough Sets Approach ([8]) is based on a learning by example theory, it has been used as a methodological tool for handling vagueness, uncertainty and noise in the collected data. With respect to the Teamwork example used in this paper we identified the stability of the team composition (variation factor 6 in figure 6) as being the core attribute for explaining the performance of the teams with respect to the team spirit (quality focus attribute 3 in figure 6).

Figure 6: GQM Abstraction sheet - Teamwork Effectiveness Model

<table>
<tr><th>Goal</th><th>Object</th><th>Purpose</th><th>Quality Focus</th><th>Viewpoint</th><th>Context</th></tr>
<tr><td>STG1.3</td><td>teamwork</td><td>baselining</td><td>effectiveness</td><td>PICT</td><td><i>Project X</i></td></tr>
<tr><td colspan="2"><b>Quality Focus</b>
Effectiveness of teamwork:
1. degree of compliance to team plans (effort, quality of team deliverables, adherence to internal teamwork processes)
2. spreading of competence
3. team spirit</td><td colspan="4"><b>Impact on Quality Focus (variation factors)</b>
1. previous teamwork experience
2. suitability of defined process for teamwork
3. team size
4. % of time devoted to teamwork
5. balance of competence within the team
6. stability of team composition
7. degree of freedom of team to develop own plans</td></tr>
<tr><td colspan="2"><b>Baseline Hypothesis</b>[a]
1. current degree of compliance to team plans =?
2. current spread of competence =?
3. current level of spirit=?</td><td colspan="4"><b>Impact on Baseline Hypothesis</b>
1. a lot of experience in working in teams increases the team effectiveness
2. inappropriate practices and processes reduces effectiveness of teamwork
3. inappropriate team size decreases effectiveness of teamwork
4. reduced time devoted for teamwork reduce team effectiveness
5. a good mix of skills is necessary for effective teamwork and to spread competence within the organization
6. frequent changes of membership in the team reduce team effectiveness
7. empowering teams to do their own planning increases team commitment to those plans, which in turn increases the chances of compliance to the plans</td></tr>
</table>

[a] The actual values were unknown, therefore the assumed values were stated and validated as shown in figure 8.

Main constraints during the analysis task were

- the lack of an underlying descriptive software process model, leading to uncertainty in the reliability of data collected,
- the strict goal orientation during analysis, and
- the inherent characteristics of software engineering data in general ([3]).

The main purpose of this measurement programme was 'baselining'. This implies less importance related to the hypothesis validation of the variation factors and focus more on the validation of the baseline hypothesis (figure 8). But, because the purpose will change to 'control', the GQM models will evolve and validation will become a key issue. The ultimate goal for measurement must be 'improvement'.

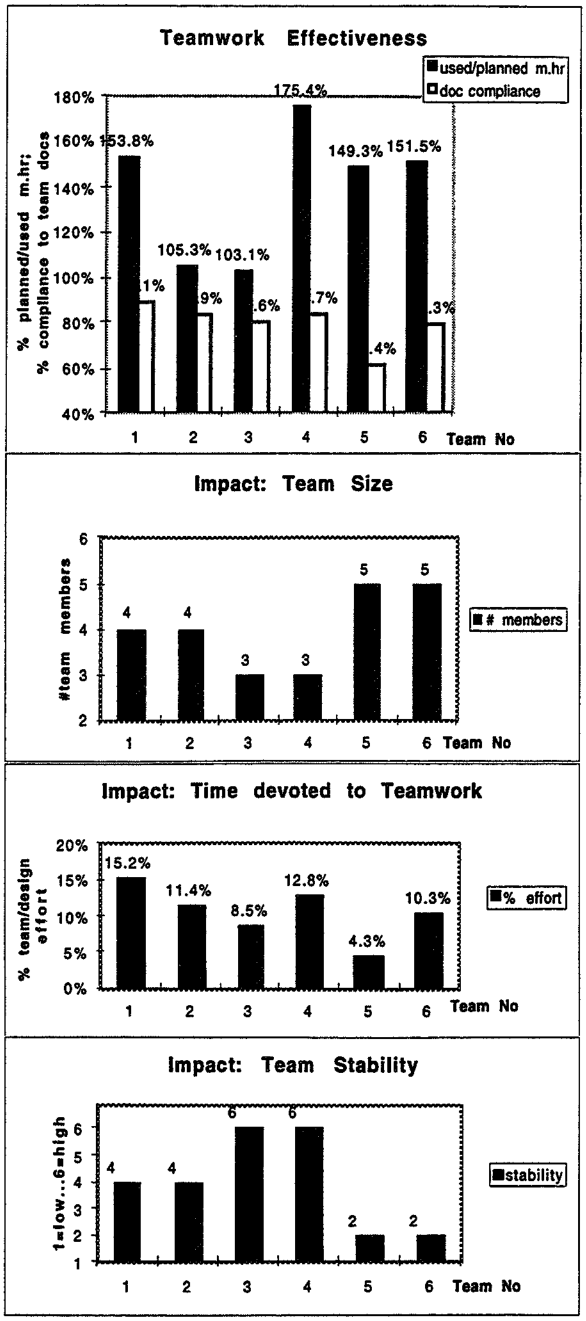
Improvement Opportunities

Three main sources for improvements of the software development process could be identified through

- the analysis results from the measurement goals, i.e., quantitative understanding,
- the GQM modelling task itself, i.e., qualitative understanding, and
- the enactment of goal-oriented measurement programme, i.e., analysis and trace of execution problems.

They uncover problems with the actual software development process, the software products delivered and the management of the software projects.

Figure 7: Validation of Variation hypothesis - Teamwork Effectiveness Model



Conclusion

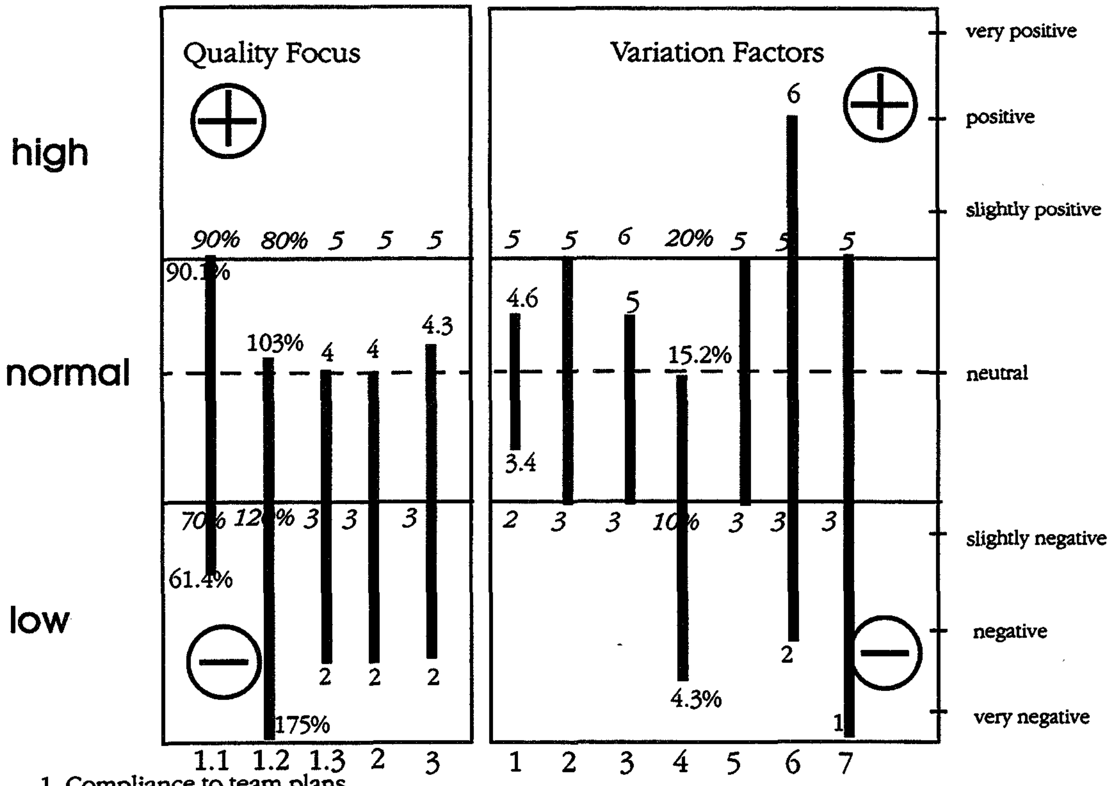
In this project we have attempted to apply systematic process improvement in a large multi-site software development. The results have been mixed but more positive than negative. The final proof being that most of the innovations are continuing in subsequent projects, albeit in some cases in a slightly modified form.

The organization set up to pursue the improvement programme is continuing although with a slightly modified composition, and is gradually starting to assume the role of 'keeper of the process experience base' and 'Strategic Improvement Management' [1] group. Although the PERFECT Approach for Improving Software Processes [1] provided a conceptual framework and useful reference model, we had difficulty in really putting it into practice. Although the underlying ideas were well-established, the practical details of the method were still evolving during the time of this project and practical experience of their use were not available. Real-world examples are needed for guidance.

The emphasis on process coaching to support technology transfer and as a way of raising process adherence has raised awareness of process issues in the organization, even if it has not yet resulted in a noticeable increase in process adherence.

The main emphasis in the project was on the application of goal-oriented measurement and the creation of a quantitative process baseline. The project was largely successful in both of these areas and the same GQM models are continuing to be used in two follow-up projects with slight modification. The main lesson learned from this first round of measurement is the need to start small and build up as the organization's measurement maturity grows. Despite having known this at the start, we still ended up with a measurement plan that was too ambitious and severely taxed the ability of the sites to collect and report data accurately and in a timely fashion. This was perhaps an inevitable consequence of the global project-wide scope of the measurement programme. Current measurement programmes are being more narrowly focussed on specific process areas. The area where we have had to be most innovative is analysis and interpretation of the results and presentation of these to project personnel. Meaningful presentation models are needed to reveal trends in the data and impacts between the variation factors and the quality focus. The two diagram types for validation of the 'Variation hypothesis' and 'Baseline hypothesis' worked well but more needs to be done. The use of Web technology to disseminate results will help in motivating the measurement activities and in the feedback of results. Finally, even in this measurement round, some useful insights have been gained into aspects of the process that were not previously understood and this has led to corrective actions in subsequent projects and this is ultimately what justifies continuation of the investment in measurement.

Figure 8: Validation of Baseline hypothesis - Teamwork Effectiveness Model



1. Compliance to team plans
   1.1 weekly document status %
   1.2 effort %
   1.3 team process adherence
2. Competence spread
3. Team spirit

1. Teamwork experience
2. Suitability of team process
3. Team size
4. Teamwork effort %
5. Competence balance
6. Composition stability

References

1 PERFECT Consortium, D22A. 'The PERFECT Approach for Improving Software Processes'. ESPRIT Project No. 9090, 1994.

2 V. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments", IEEE Transactions on Software Engineering, 14 (6), pages 758-773, June 1988.

3 L. Briand, V. Basili, and W. Thomas, "A pattern recognition approach for software engineering data analysis", IEEE Trans. Software Eng., vol. 18, no. 11, Nov. 1992

4 Q-Labs, "How could Rough Sets be used for GQM-based Data analysis", KL/QLS 96:0354, October 1996 (unpublished).

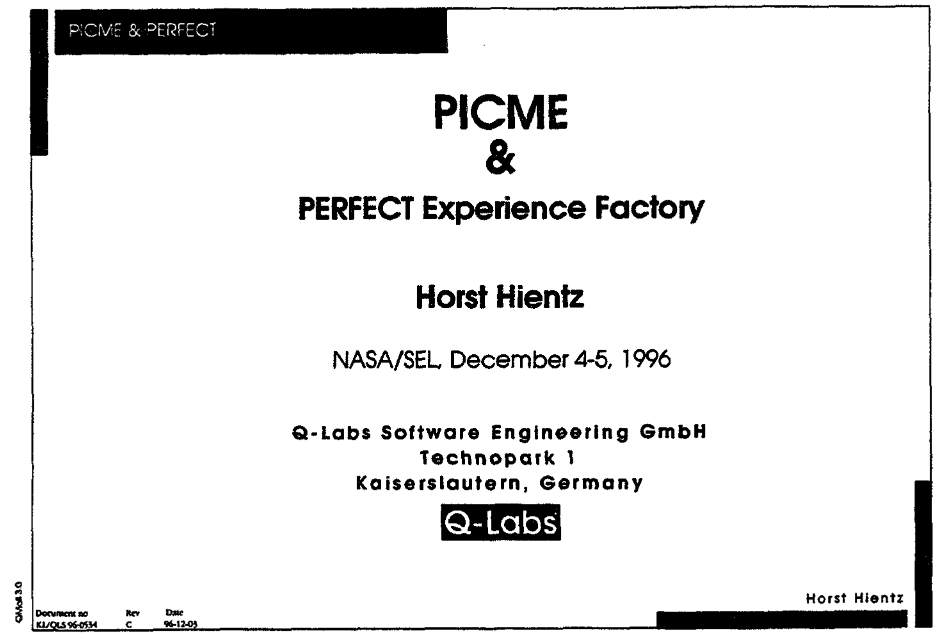
5 Victor R. Basili and H. Dieter Rombach. TAME: Integrating measurement into software environments. Technical Report CS-TR-1764 and TAME-TR-1-1987, Department of Computer Science, University of Maryland, College Park, MD 20742, June 1987.

6 Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience Factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

7 L. Briand, V. Basili, and W. Thomas, A pattern recognition approach for software engineering data analysis, IEEE Trans. Software Eng., vol. 18, no. 11, Nov. 1992.

8 J.W. Grzymala-Busse, LERS - A system for learning from examples based on rough sets. In. R. Slowinsky, editor, *Intelligent Decision Support: Handbook of Applications and Advances of Rough Sets Theory*, Kluwer Academic Publ, 1992.

9 Lionel Briand, Christiane Differding, and H. Dieter Rombach, Practical Guidelines for Measurement-Based Process Improvement, Published as Technical Report for the International Software Engineering Research Network (ISERN-96-05), 1996.



PICME & PERFECT

PICME
&
PERFECT Experience Factory

Horst Hientz

NASA/SEL, December 4-5, 1996

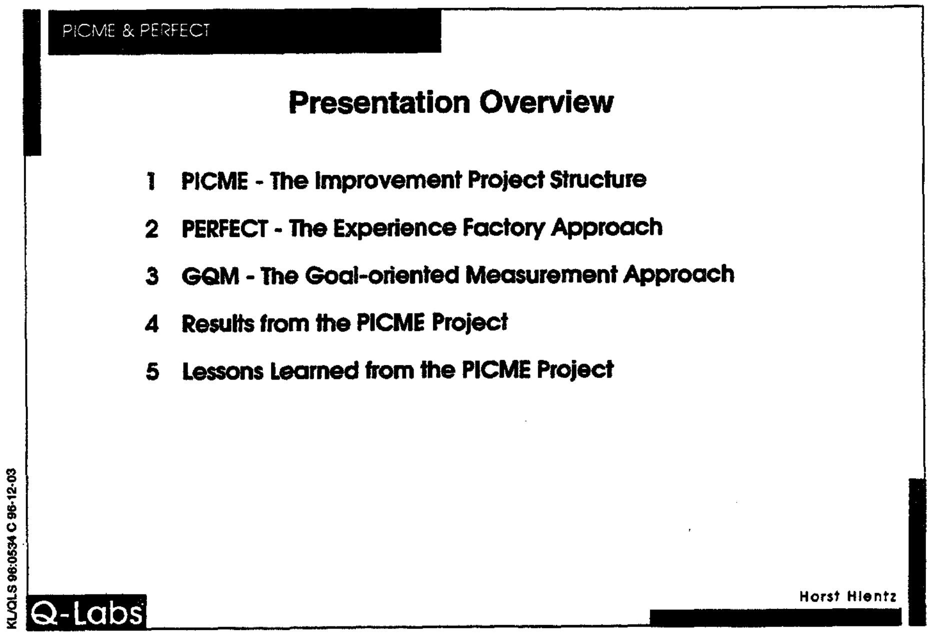
Q-Labs Software Engineering GmbH
Technopark 1
Kaiserslautern, Germany

Q-Labs

Horst Hientz

QMall 3.0

Document no	Rev	Date
KL/QLS 96-0534	C	96-12-03



PICME & PERFECT

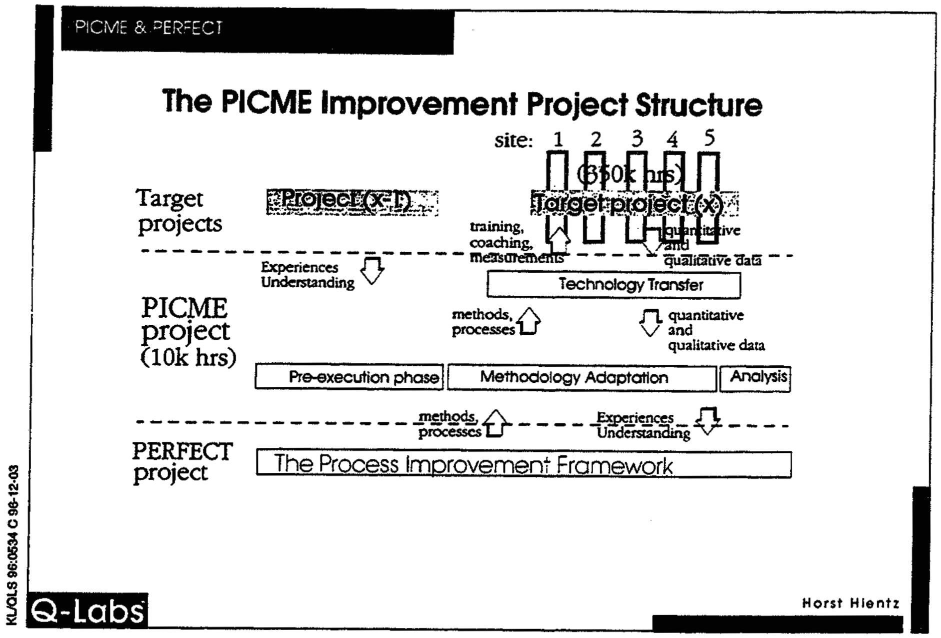
Presentation Overview

1 PICME - The Improvement Project Structure
2 PERFECT - The Experience Factory Approach
3 GQM - The Goal-oriented Measurement Approach
4 Results from the PICME Project
5 Lessons Learned from the PICME Project

KL/QLS 96:0534 C 96-12-03

Q-Labs

Horst Hientz



PICME & PERFECT

The PICME Improvement Project Structure

site: 1 2 3 4 5

(350k hrs)

Target projects

Project (x-1)

Target project (x)

training, coaching, measurements

quantitative and qualitative data

Experiences Understanding

Technology Transfer

PICME project (10k hrs)

methods, processes

quantitative and qualitative data

Pre-execution phase

Methodology Adaptation

Analysis

methods, processes

Experiences Understanding

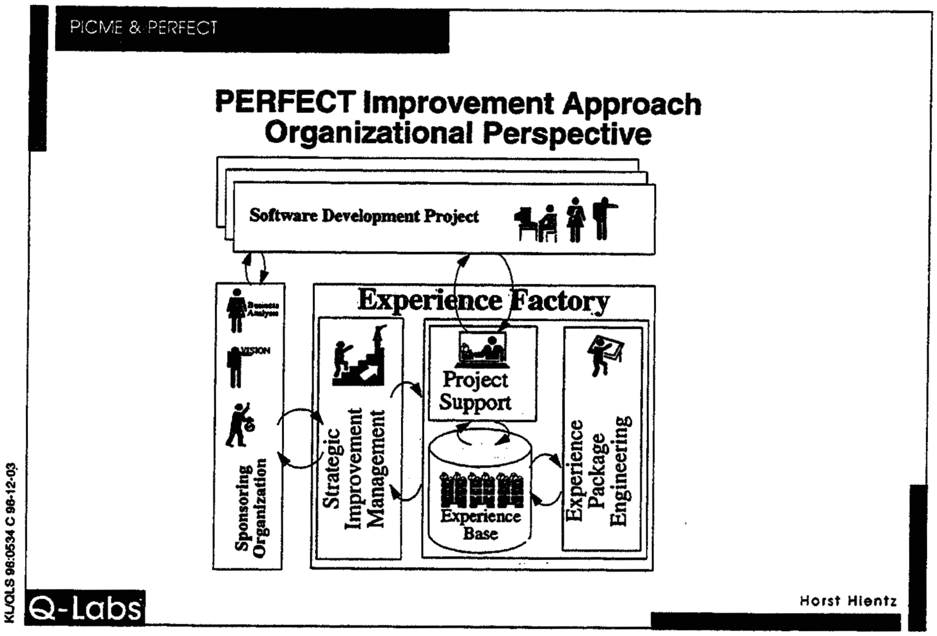
PERFECT project

The Process Improvement Framework

KL/QLS 96:0534 C 96-12-03

Q-Labs

Horst Hientz



PICME & PERFECT

PERFECT Improvement Approach Organizational Perspective

Software Development Project

Experience Factory

Sponsoring Organization

Strategic Improvement Management

Project Support

Experience Base

Experience Package Engineering

KL/QLS 96:0534 C 96-12-03

Q-Labs

Horst Hientz

PICME & PERFECT

PICME usage of PERFECT Experience Factory entities

- **Strategic Improvement Management**
  - Continues Software Process Improvement and New Technology Selection
- **Experience Package Engineering**
  - Experience generalization
  - Organization Processes (for SW development and measurement) enhancement
- **Project Support**
  - Project analysis and understanding
  - Project guidance
  - GQM-based measurement programme application

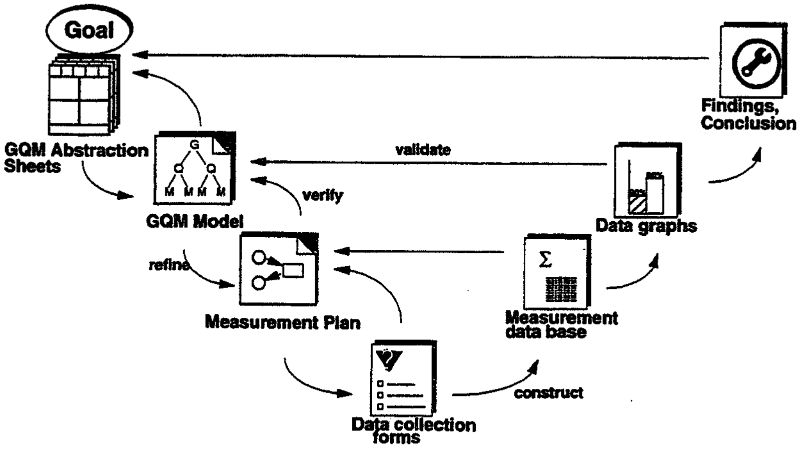
KL/QLS 96:0534 C 96-12-03

Q-Labs

Horst Hientz

PICME & PERFECT

GQM - V Model



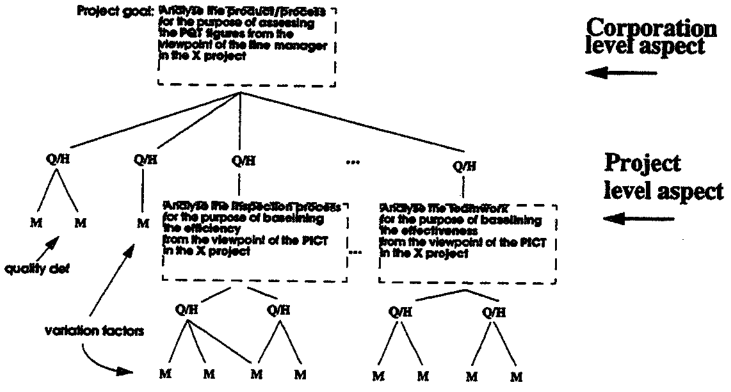
KL/QLS 96:0534 C 96-12-03

Q-Labs

Horst Hientz

PICME & PERFECT

GQM Model 'Goal tree'



Project goal: Analyse the product/process for the purpose of assessing the PQT figures from the viewpoint of the line manager in the X project

Analyse the inspection process for the purpose of baselining the efficiency from the viewpoint of the PICT in the X project

Analyse the teamwork for the purpose of baselining the effectiveness from the viewpoint of the PICT in the X project

Q-Labs

Horst Hientz

KL/QLS 96:0534 C 96-12-03

PICME & PERFECT

GQM - Abstraction Sheet

Goal	Object	Purpose	Quality Focus	View-point	Context
STG1.3	teamwork	baselining	effectiveness	PICT	*Project X*

Quality Focus

Effectiveness of teamwork:

1. degree of compliance to team plans (effort, quality of team deliverables, adherence to internal team-work processes)
2. spreading of competence
3. team spirit

Impact on Quality Focus (variation factors)

1. previous teamwork experience
2. suitability of defined process for teamwork
3. team size
4. % of time devoted to teamwork
5. balance of competence within the team
6. stability of team composition
7. degree of freedom of team to develop own plans

Baseline Hypothesis

1. current degree of compliance to team plans =?
2. current spread of competence =?
3. current level of spirit=?

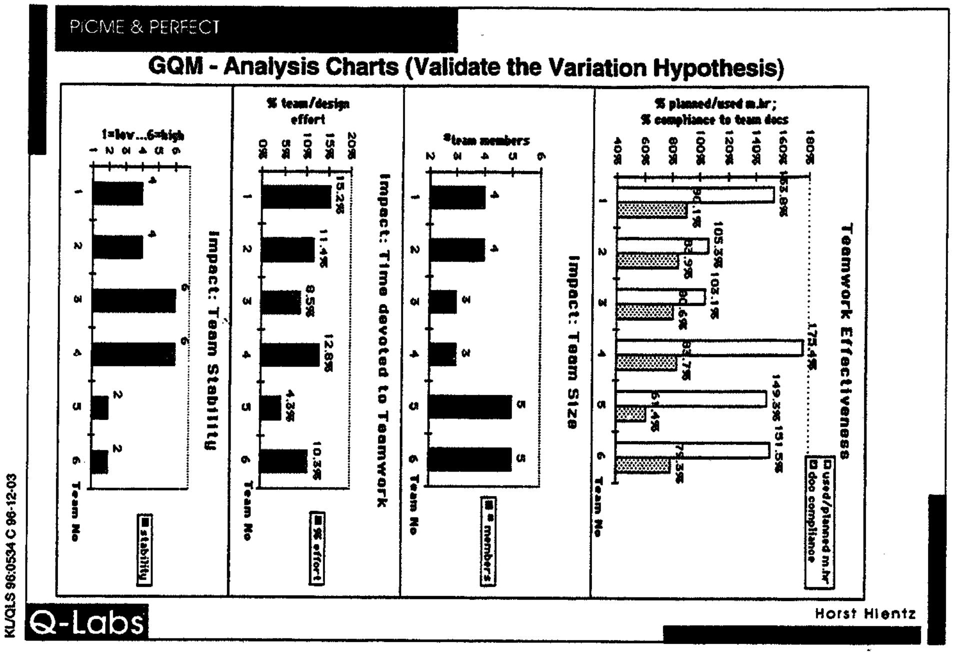
Impact on Baseline Hypothesis

1. a lot of experience in working in teams increases the team effectiveness
2. inappropriate practices and processes reduces effectiveness of teamwork
3. inappropriate team size decreases effectiveness of teamwork
4. reduced time devoted for teamwork reduce team effectiveness
5. a good mix of skills is necessary for effective teamwork and to spread competence within the organization
6. frequent changes of membership in the team reduce team effectiveness
7. empowering teams to do their own planning increases team commitment to those plans, which in turn increases the chances of compliance to the plans

Q-Labs

Horst Hientz

KL/QLS 96:0534 C 96-12-03



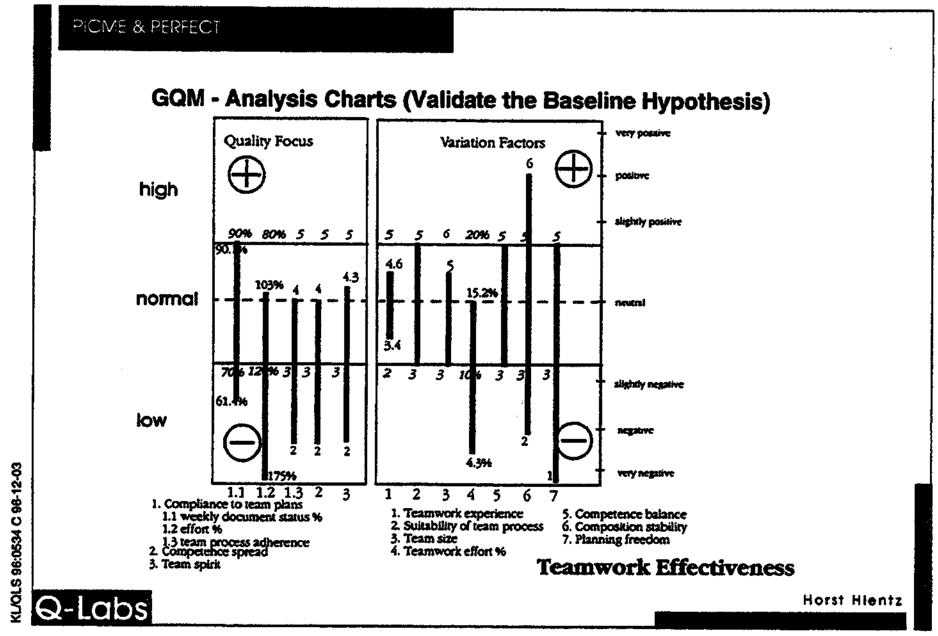
PICME & PERFECT

GQM - Analysis Charts (Validate the Variation Hypothesis)

Horst Hientz

Q-Labs

KL/QLS 96:0534 C 96-12-03



PICME & PERFECT

GQM - Analysis Charts (Validate the Baseline Hypothesis)

1. Compliance to team plans
   1.1 weekly document status %
   1.2 effort %
   1.3 team process adherence
2. Competence spread
3. Team spirit

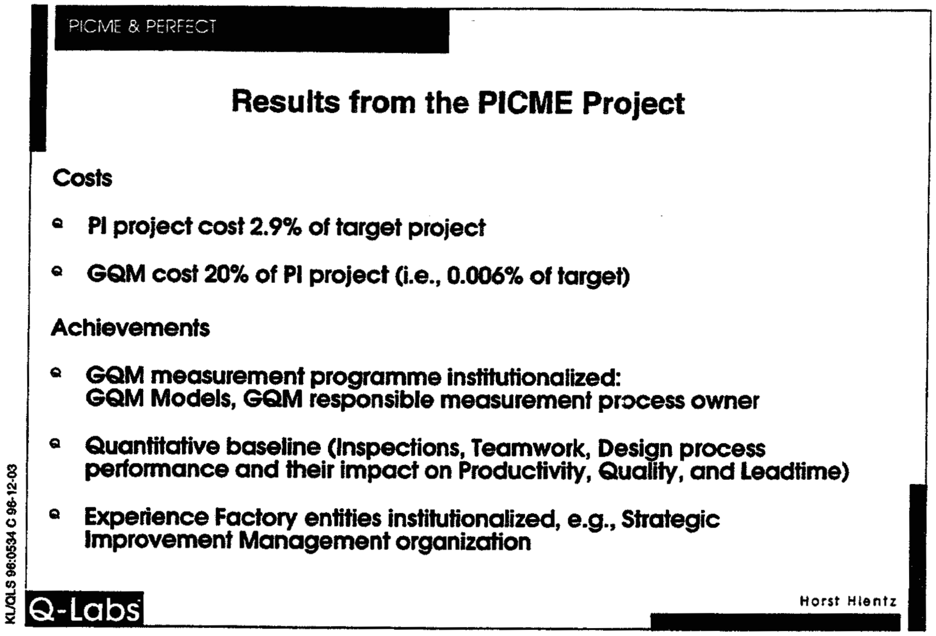
1. Teamwork experience
2. Suitability of team process
3. Team size
4. Teamwork effort %
5. Competence balance
6. Composition stability
7. Planning freedom

Teamwork Effectiveness

Horst Hientz

Q-Labs

KL/QLS 96:0534 C 96-12-03



PICME & PERFECT

Results from the PICME Project

Costs

- PI project cost 2.9% of target project
- GQM cost 20% of PI project (i.e., 0.006% of target)

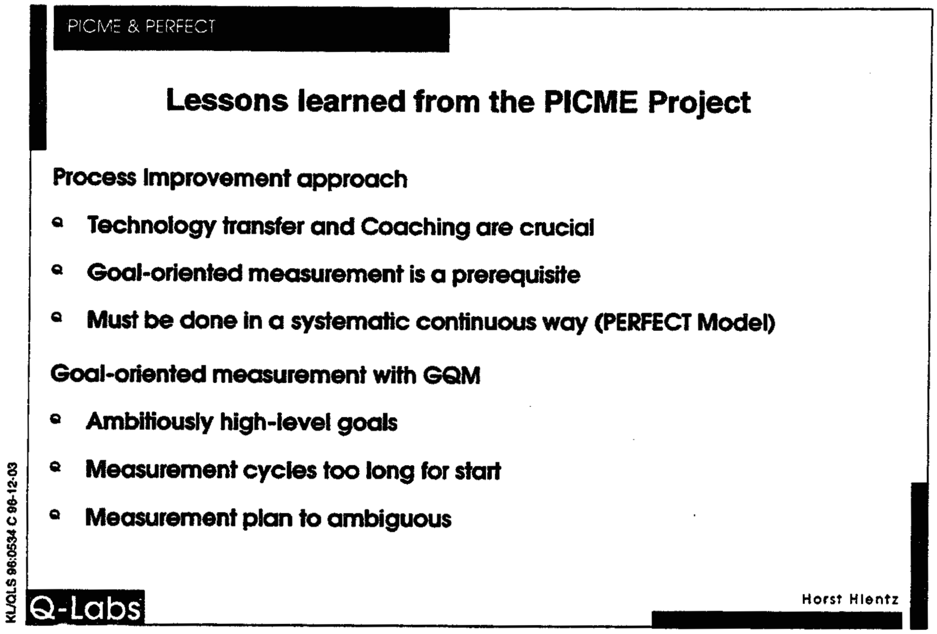
Achievements

- GQM measurement programme institutionalized:
GQM Models, GQM responsible measurement process owner
- Quantitative baseline (Inspections, Teamwork, Design process performance and their impact on Productivity, Quality, and Leadtime)
- Experience Factory entities institutionalized, e.g., Strategic Improvement Management organization

KL/QLS 96:0534 C 96-12-03

Q-Labs

Horst Hientz



PICME & PERFECT

Lessons learned from the PICME Project

Process Improvement approach

- Technology transfer and Coaching are crucial
- Goal-oriented measurement is a prerequisite
- Must be done in a systematic continuous way (PERFECT Model)

Goal-oriented measurement with GQM

- Ambitiously high-level goals
- Measurement cycles too long for start
- Measurement plan to ambiguous

KL/QLS 96:0534 C 96-12-03

Q-Labs

Horst Hientz

An Empirical Study of Process Conformance

Sivert Sørumgård[1]
Norwegian University of Science and Technology
3 December 1996

S6-61
415649
360698
p18

Abstract

An experiment was carried out at the Norwegian University of Science and Technology (NTNU) in order to investigate a concept called *process conformance*. The experiment was based on previous experiments carried out at the Software Engineering Laboratory (SEL) (Basili, 1996). The purpose of the experiment was to compare two variants of a process and see whether the type of process had an impact on the level of process conformance. Another goal was to investigate the correlation between the degree of conformance and deviation in product quality. The results obtained so far indicate some evidence that process conformance and product quality deviation are correlated. The process type had no apparent impact on conformance.

1.0 Introduction

Experiences from an experiment carried out in the context of SEL at the University of Maryland (UMD) in 1995 suggested that one problem concerning experiments with software engineering processes is the question of whether the process under investigation is actually used by the subjects in the experiment - i.e. *process conformance* (Basili, 1996). Hence, we define process conformance as:

> *The degree of agreement between a process definition and the process that is actually carried out.*

As we consider the definition above, three problems immediately arise: How to measure the *degree of agreement*, to define in more detail what *agreement* means, and finally what is meant by a *process definition*. Here, we will put emphasis on the first problem - how to measure process conformance.

Some related work has been proposed (e.g. Cook, 1994; Cugola, 1995; Miyazaki, 1987; etc.), but not in the domain of software process experiments, which tend to study low-level and thought-intensive processes. Thus, the current approaches, which are mostly focused on higher-level processes, were not considered appropriate.

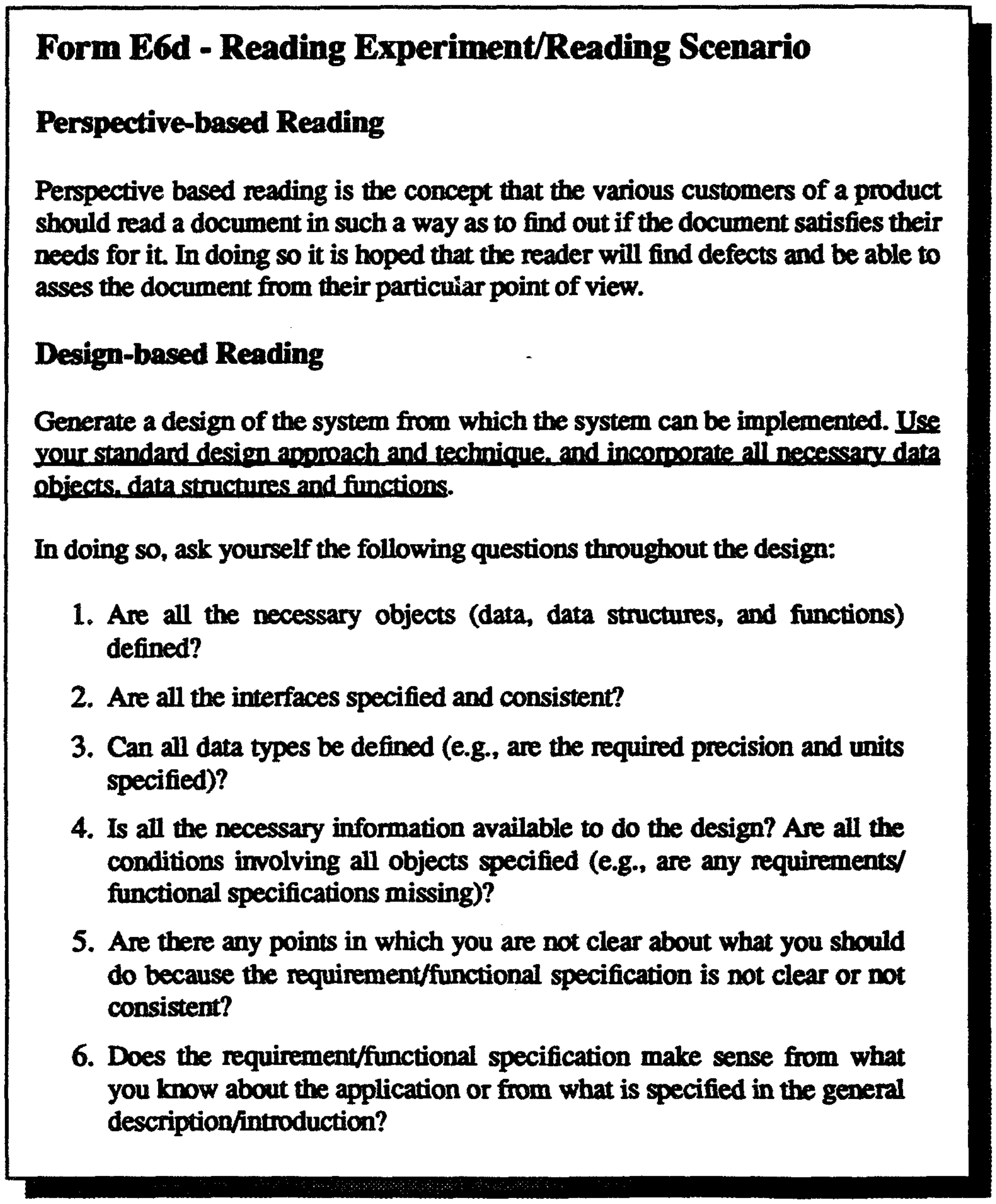
This paper is describing an experiment that was carried out to investigate process conformance. First, the context of the experiment, the goal and hypotheses, and its design are described. Then the conformance measurement is explained. Further, the required preparations and the execution of the experiment are outlined briefly, before the experimental results are presented. Finally, a conclusion summarizes the experiment and presents some ideas about the possible future direction of this work.

[1] Complete address: Division of Computer Science, Norwegian University of Science and Technology, N-7034 Trondheim, Norway
Email: sivert@idt.ntnu.no, phone: +47 73 59 44 79, fax: +47 73 59 44 66.

2.0 Context of the Experiment

The goal of the experiment was to investigate process conformance. In particular, we wanted to compare two variants of the same process where one variant had been modified in order to make it more *explicit* as well as requiring that the subjects delivered *intermediate results*. Modifications such as these result in a process which is defined in more detail, and thus may be expected to be easier to follow correctly since the room for interpretetaion is reduced. The process we used was *Perspective-Based Reading* (Basili, 1996) as applied in the UMD experiment referred to earlier (Basili, 1996).

PBR is a technique for reading requirements specifications in order to find defects. The idea is that people read it from three different perspectives: Design, Use, and Test. In our experiment, we applied only the design perspective in order to reduce the number of variables.



Form E6d - Reading Experiment/Reading Scenario

Perspective-based Reading

Perspective based reading is the concept that the various customers of a product should read a document in such a way as to find out if the document satisfies their needs for it. In doing so it is hoped that the reader will find defects and be able to asses the document from their particular point of view.

Design-based Reading

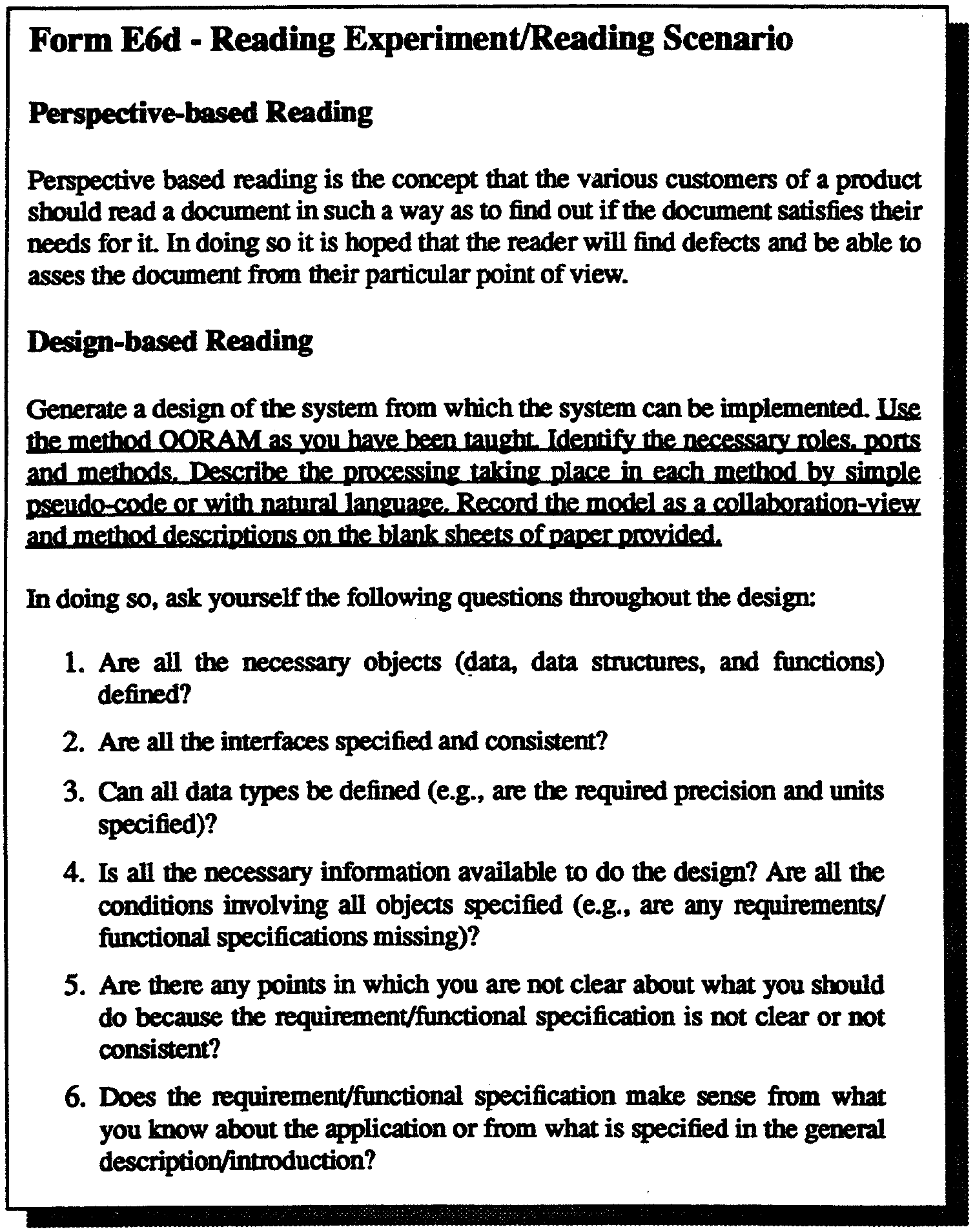
Generate a design of the system from which the system can be implemented. Use your standard design approach and technique, and incorporate all necessary data objects, data structures and functions.

In doing so, ask yourself the following questions throughout the design:

1. Are all the necessary objects (data, data structures, and functions) defined?
2. Are all the interfaces specified and consistent?
3. Can all data types be defined (e.g., are the required precision and units specified)?
4. Is all the necessary information available to do the design? Are all the conditions involving all objects specified (e.g., are any requirements/functional specifications missing)?
5. Are there any points in which you are not clear about what you should do because the requirement/functional specification is not clear or not consistent?
6. Does the requirement/functional specification make sense from what you know about the application or from what is specified in the general description/introduction?

Figure 1. Process description - PBR.

The design perspective, as described by PBR, is characterized by a short description and a set of questions. The "designer" is to apply a design technique and make a design for the system, and during this process he is to apply the questions in order to identify defects. However, the description puts forward no requirements as to which design technique is to be applied. The modified version of PBR was made more explicit by requiring a specific design technique called *OOram* (Object Oriented Role Analysis Method) (Reenskaug, 1995) to be applied. Another modification was to require the subjects to deliver their design as an intermediate result of the process. Hence, variation in process execution could be assumed to be reduced. The process descriptions for the unmodified and modified versions are provided in Figure 1 and Figure 2 respectively.



Form E6d - Reading Experiment/Reading Scenario

Perspective-based Reading

Perspective based reading is the concept that the various customers of a product should read a document in such a way as to find out if the document satisfies their needs for it. In doing so it is hoped that the reader will find defects and be able to asses the document from their particular point of view.

Design-based Reading

Generate a design of the system from which the system can be implemented. Use the method OORAM as you have been taught. Identify the necessary roles, ports and methods. Describe the processing taking place in each method by simple pseudo-code or with natural language. Record the model as a collaboration-view and method descriptions on the blank sheets of paper provided.

In doing so, ask yourself the following questions throughout the design:

1. Are all the necessary objects (data, data structures, and functions) defined?
2. Are all the interfaces specified and consistent?
3. Can all data types be defined (e.g., are the required precision and units specified)?
4. Is all the necessary information available to do the design? Are all the conditions involving all objects specified (e.g., are any requirements/ functional specifications missing)?
5. Are there any points in which you are not clear about what you should do because the requirement/functional specification is not clear or not consistent?
6. Does the requirement/functional specification make sense from what you know about the application or from what is specified in the general description/introduction?

Figure 2. Process description - MPBR.

2.1 Hypotheses

The experiment was focused on comparing and measuring the degree of conformance. Thus we assume that the two process variants will be different as far as the conformance is concerned. This means, we assume that people tend to follow one process variant more closely than the other. Based on this, the hypothesis and its associated null hypothesis for this experiment was:

H1 Subjects applying the modified version of PBR will show a higher degree of process conformance than subjects applying the unmodified version of PBR.

H0,1 There is no difference in process conformance between subjects applying the modified version of PBR and subjects applying the unmodified version of PBR.

There are a number of additional hypotheses that are also of high interest in the context of process conformance. In this paper, we will also consider the following hypothesis and associated null hypothesis:

H2 There is no correlation between process conformance and deviation in product quality.

H0,2 Process conformance and deviation in product quality are associated variables.

In the following discussion, the modified version of Perspective-Based Reading will be referred to as *MPBR*, while the unmodified version is labelled *PBR*.

2.2 Measuring Process Conformance

In order to test our hypotheses, we need a way of measuring process conformance and deviation in product quality. One way of doing this would be to observe how the process was carried out and then compare these observations with the process description. This can be accomplished by collecting a number of observations for each subject's process execution, e.g. time used, and product size and quality, and compare these observations with *predicted* values. Or, alternatively, the sample means may be substituted for the predicted values, if we assume that the average observations represent a typical process execution.

Based on the set of observations for each subject, we can construct a *deviation vector,* which is a model of how the process execution diverges from the expected performance. A deviation vector with two dimensions, time and quality, is depicted in Figure 2. Here, the predicted execution of process i is represented by a vector P_i, while the actual execution is represented by the vector E_i. The deviation vector is now defined as the difference between the predicted and actual execution, where the value in each dimension is the unsigned difference between prediction and execution. Thus, the deviation vector in Figure 2 becomes $[\ |T_{i,e} - T_{i,p}|, |Q_{i,e} - Q_{i,p}|\ ]$.

The conformance *measurement* can now be defined as the *length* of this vector when all the dimensions have been normalized by dividing the difference by the expected value so that the different dimensions can be combined. In this experiment, we used the observations *time*, *product size*, and *product quality* (these will be explained later) in the deviation vector, and thereby obtained the conformance measurement for subject i given by Equation 1, where the

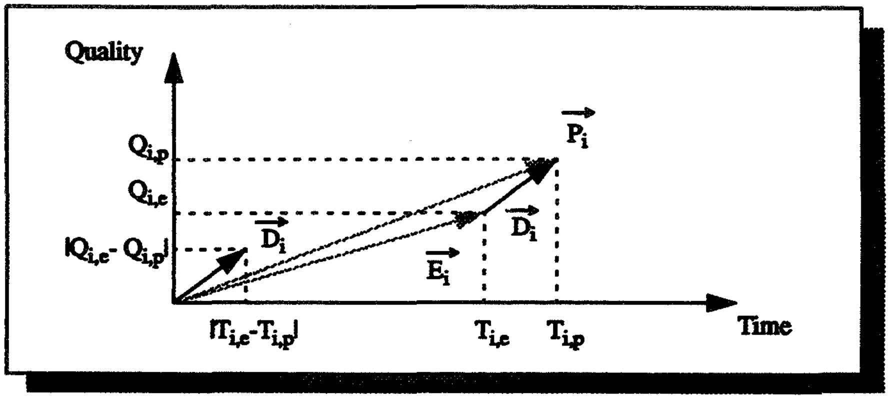


Figure 2. The positive deviation vector.

predicted values are replaced by the sample averages. This measurement was used to test hypothesis H0,1.

$$C_i = \frac{1}{\sqrt{3}}\sqrt{\left(\frac{|t_i-E(t)|}{E(t)}\right)^2+\left(\frac{|s_i-E(s)|}{E(s)}\right)^2+\left(\frac{|q_i-E(q)|}{E(q)}\right)^2} \qquad (1)$$

For testing the second hypothesis, we also need a measure of product quality deviation. However, this is exactly the third dimension in the vector above, i.e. product quality deviation for subject *i* is defined by

$$dQ_i = \frac{|q_i-E(q)|}{E(q)} \qquad (2)$$

Testing the association between the quality deviation given by Equation 2 and the process conformance measurement given by Equation 1 is not reasonable to do since quality deviation is also a component in process conformance. Thus, a simplified conformance measurement is required for testing the second null hypothesis H0,2. In this simplified measurement, we used only the two dimensions *time* and *size*, as given by the equation below.

$$C_i = \frac{1}{\sqrt{2}}\sqrt{\left(\frac{|t_i-E(t)|}{E(t)}\right)^2+\left(\frac{|s_i-E(s)|}{E(s)}\right)^2} \qquad (3)$$

2.3 Experimental Design

We used a fractional factorial design where we blocked the subjects on *document order* and *technique type* (these variables will be explained later), thus obtaining the design illustrated in Figure 3 where the actual number of subjects in each block is indicated in parenthesis. The subjects in the experiment were 48 graduate students in their last year of study before the diploma thesis. The number of subjects in each block indicated in the figure above are slightly uneven because some of the subjects that signed up for the experiment did not show up. Every subject read two software requirements specifications that were seeded with a set of known defects, and applied a specific technique in order to find the defects, using the same technique for both documents. Thus, there were three independent variables as described in Table 1 below.

Technique	PBR		Modified PBR	
Document order	ATM-PG	PG-ATM	ATM-PG	PG-ATM
First document	ATM	PG	ATM	PG
Second document	PG	ATM	PG	ATM
Number of subjects	13 (12)	12 (11)	13 (12)	13 (13)
	Group 1	Group 2	Group 3	Group 4

Figure 3. Design of the experiment.

Variable	Scale	Unit	Comments
Technique	nominal	-	Technique has two values: PBR and MPBR.
Document	nominal	-	Document can take two values: ATM and PG.
Order	nominal	-	Order has two values: ATM-PG and PG-ATM.

Table 1. Independent variables.

The dependent variables which were collected are summarized in Table 2. The basic variables are *time*, *defects*, and *size*. The two latter had to be adjusted for difference in document "size" (size in terms of the number of seeded defects). Thus, they were replaced by *rates*. In addition to the variables described in Table 2, a simplified variant of the conformance measurement and the measurement of product quality deviation, as given by Equation 2 and 3, were also needed, as discussed previously.

Variable	Scale	Unit	Comments
Time	ratio	minute	Time was measured by the subjects themselves.
Defects	absolute	-	The number of defects the subject identified which was also on our list of known defects.
Size	absolute	-	Total number of potential defects identified by the subject.
Defect detection rate	ratio	-	Number of real defects identified divided by the total number of defects in the document.
Adjusted size	ratio	-	Number of assumed defects (size) divided by the total number of defects in the document.
Conformance	ordinal	-	Conformance measured using time, defect detection rate, and size as parameters.

Table 2. Dependent variables.

2.4 Preparation and Execution of the Experiment

This experiment was based on the UMD experiment referred to earlier, and much of the experimental material, documents and forms were reused, in addition to the process of *Perspective-Based Reading* as explained earlier. The two documents that were read by the subjects were completely unchanged from the UMD versions, and were:

- A specification for an automated teller machine network, called the *ATM* document. The latest version available as of 19th April 1996 was used. The document was 16 pages long and contained 29 seeded defects.
- A specification for a parking garage control system, called the *PG* document. Again, the latest version available at the time of the experiment was used. This document was 17 pages long and contained 27 seeded defects.

The defect lists applied were also the same as in the UMD experiment. As for the forms applied, we only used one type of form for all subjects regardless of process type. We could do this because we only applied one perspective, while the UMD experiment investigated all the three perspectives of PBR.

Since there was no pretest of the subjects, they were assigned to the blocks randomly. The subjects were split into two separate groups when they received orientation and training in front of the experiment. They were not told about the hypothesis or about the differences in the processes. All subjects received the same type and amount of training. After the training session, which was one hour for each of the two groups, the subjects had one week to read the documents and mark the defects. However, they were instructed not to use more than 1:45 hours on each document. When the subjects had read both documents, they returned them to us for scoring. Two persons scored the documents independently, and then resolved any conflicts by discussing each disagreement.

Finally, we removed the outliers from the data set. First, three subjects that failed to show up were removed. Next, those reporting no effort spent, i.e. no time used to find defects, were removed. Finally, subjects having found no defects, even though they reported some effort spent, were removed if their documents showed no clear signs of being read.

3.0 Results of the Experiment

In order to test the first hypothesis, i.e. whether the process modifications caused improved process conformance, the eight samples were first compared all at once to determine if they could all be assumed to come from the same population. Here, the *Kruskal-Wallis* test was used (Siegel, 1988). The test indicated that the null hypothesis could not be rejected (p=0.5), meaning that there were no significant differences between any of the eight samples, and thus technique could not be considered to have any effect on process conformance. This was confirmed by grouping subjects using the same technique into two samples and test the difference by using the *Wilcoxon-Mann-Whitney* test (Siegel, 1988). The null hypothesis for this test was that the two samples were drawn from the same population, while the alternative hypothesis was that the sample using MPBR scored lower on the conformance measurement. The null hypothesis was not rejected (p=0.39). The medians for process conformance for the eight samples are shown in the chart in Figure 4, and illustrate the similarity between the samples.

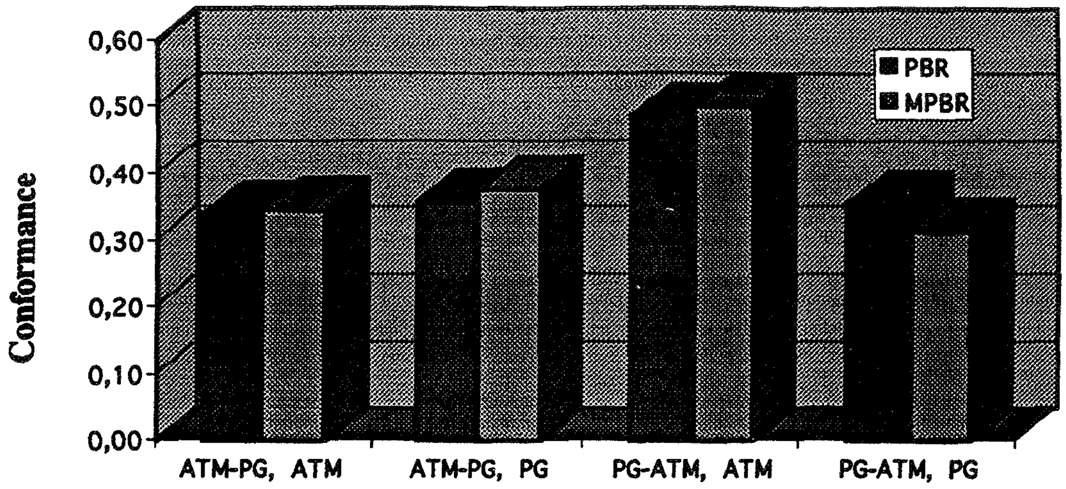


Figure 4. Effect of process modifications on conformance median.

The second hypothesis was focused on the association between the simplified measure of process conformance, as given by Equation 3, and deviation in product quality, as measured by Equation 2. The assumption was that subjects who were not following the process correctly, as indicated by a ***high*** deviation value, would *not* deliver a product that was close to the average of the sample. This is the principle which many process improvement approaches are based on, that by reducing the variance in the process execution, a more stable process performance is ensured.

To test this hypothesis, the *Spearman rank-order correlation coefficient* (Siegel, 1988) was computed and used to decide whether the null hypothesis, which suggests that the two samples are not correlated, could be rejected. The rejection was confirmed by the test (p=0.0010), meaning that with a significance level of a=0.05, the two variables can be considered significantly associated. The two variables are plotted in Figure 5.

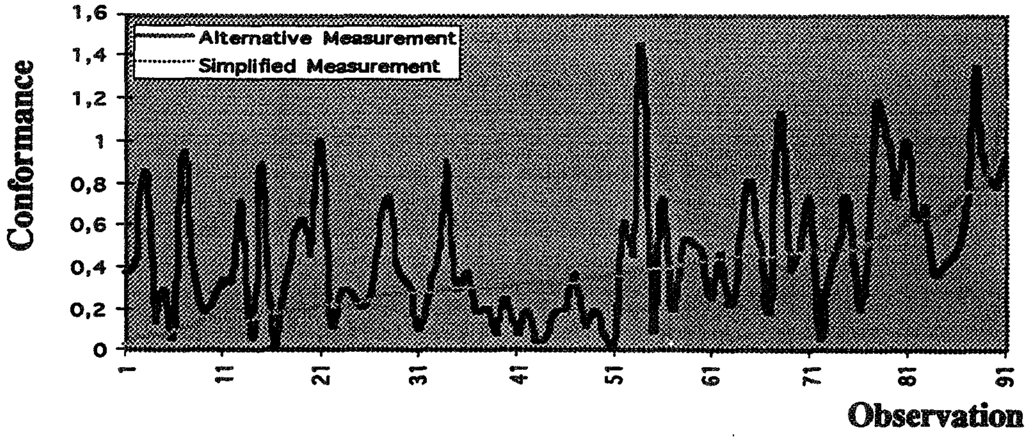


Figure 5. Association between conformance and quality deviation.

4.0 Conclusion

Based on the preceding tests, we can conclude that the suggested modifications assumed to increase the level of process conformance had no effect. However, a significant level of correlation was detected between process conformance, as measured by the simplified measurement given by Equation 3, and the deviation in product quality, as given by Equation 2.

The problem with this experiment is that the subjects used, since being students, can not be considered representative for the population of professional programmers. Especially considering that the experiment was carried out as a compulsory assignment. The consequence of the experimental situation could be that subjects being assigned the modified version of PBR developed reactive effects due to the presumably high work-load of also delivering an intermediate product. Thus, we have a potential interaction effect between the treatment and the sample, combined with possible reactive effects due to the experimental environment, meaning that external validity may be compromized.

However, in the case of the association between process conformance and deviation in product quality, the threat might be less relevant since we are essentially comparing two kinds of deviations. However, whether the two variables are significantly correlated also in other populations and environments can only be determined empirically.

This paper approached process conformance from an experimental point of view - i.e. we considered lack of conformance a problem in software process experiments. however, this is a problem also in other contexts. One of the major problems in software development is lack of predictability - this problem may be reduced by achieving a more stable product quality through controlling process conformance. Proper process conformance is also necessaru to reuse experiences effectively both within one organization as well as in different organizations. Thus, process conformance may be considered an important aspect of *process quality.*

In the experiment described here, we attempted to influence process conformance by modifying the process. However, we can imagine various other ways of influencing conformance, e.g by education and training, or by control and enforcement. The way of improving process conformance must be related to the context in each case. Different ways may be beneficial in e.g. an experiment context than in a development context.

References

(Basili, 1996) Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sørumgård and Marvin Zelkowitz. *The Empirical Investigation of Perspective-Based Reading*. To appear in the Journal of Empirical Software Engineering.

(Cook, 1995) Jonathan E. Cook and Alexander L. Wolf. *Automating Process Discovery through Event-Data Analysis*. In Proc. ICSE 17, ACM, 1995.

(Cugola, 1995) G. Cugola, E. Di Nitto, C. Ghezzi and M. Mantione. *How To Deal With Deviations During Process Model Enactment*. In Proc. International Conference on Software Engineering, ACM, 1995.

(Miyazaki, 1987) Yukio Miyazaki and Noritoshi Mutakami. *Software Metrics Using Deviation Value*. In Proc. International Conference on Software Engineering, ACM Press, 1987.

(Reenskaug, 1995) Trygve Reenskaug et. al. *Working with Objects - The OOram Way to Software Success*. Mannerheim Publications/Prentice Hall, October 1995.

(Siegel, 1988) Sidney Siegel and N. John Castellan, Jr. *Nonparametric Statistics*. Second edition, McGraw-Hill, 1988.

An Empirical Study of Process Conformance

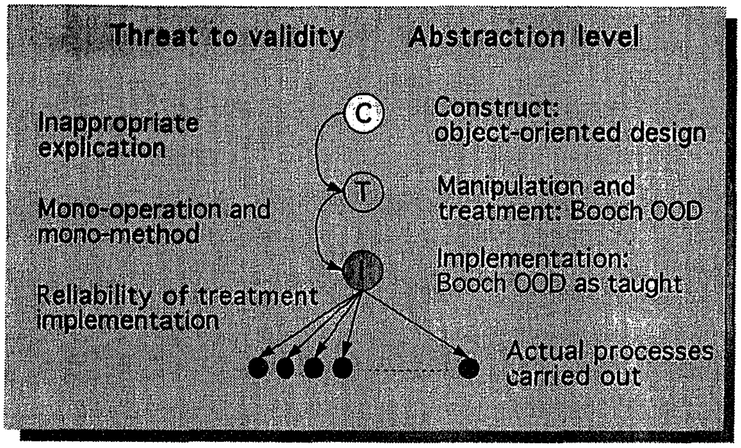
Sivert Sørumgård
Norwegian University of Science and Technology

Contents

- Informal definition, importance within experiments.
- Conformance as deviations - the deviation vector.
- Measuring process conformance.
- Process modifications.
- An experiment to investigate process conformance.
- Variables and hypotheses.
- Results from the experiment.
- Effects within software process experiments.
- Effects within software development.
- Conclusion.

Experimental context is assumed.

Conformance in Experiments



Threat to validity

Abstraction level

Inappropriate explication

Construct: object-oriented design

Mono-operation and mono-method

Manipulation and treatment: Booch OOD

Reliability of treatment implementation

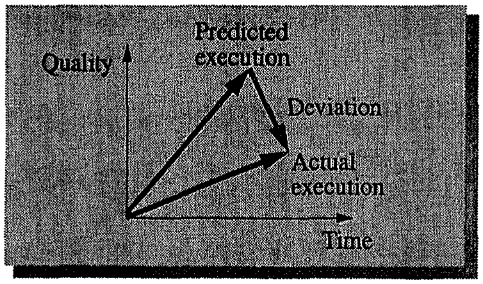
Implementation: Booch OOD as taught

Actual processes carried out

Are the processes carried out the way we think?

The Deviation Vector

- Observations indicate what's important.
- Use as dimensions in a vector - parametrized model.

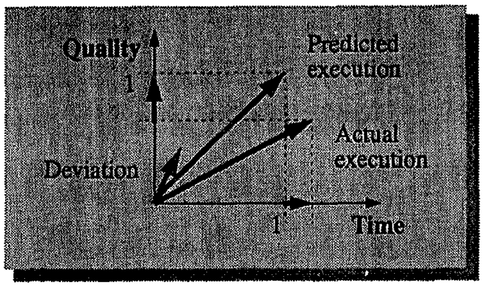


- Deviation is difference between Execution and Prediction.
- Rules for combining task deviations to obtain process deviation.

The deviation vector is a model of conformance.

Measuring Conformance

- Define measurement based on deviation vector.
- Differences as fractions, independent of scale.



- Then, dimensions in the deviation vector may be compared.
- Can define measure of process conformance.

Process conformance: Length of deviation vector.

Process Modifications

Enabling effective measurement

- Observations reflecting process characteristics.
- Intermediate products.

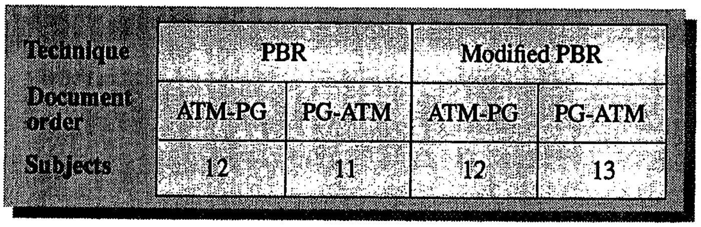
Improving conformance

- Remove ambiguities and reduce room for interpretation.
- Suggest process steps.
- Explicit and specific.
- Training, teaching, representation.

Can the process be modified to become conform?

Experimental Study

- What to compare with?
- Validity vs. usability.
- What are the relations in the empirical system?
- Two aspects: Modifications, and conformance measurement.



Technique	PBR		Modified PBR	
Document order	ATM-PG	PG-ATM	ATM-PG	PG-ATM
Subjects	12	11	12	13

Fractional factorial design, students as subjects.

Variables and Hypotheses

Variables

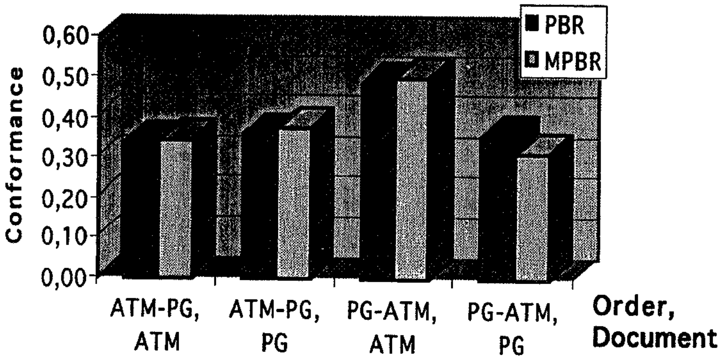
- Technique, document order, and document type.
- Measurements: Time, defect detection rate ("quality"), total number of defects found ("size"), intermediate product quality.

Hypotheses

- The modifications improve process conformance.
- The modifications lead to reduced product quality.
- The modifications lead to reduced product quality deviation.
- Conformance is associated with deviation in product quality.

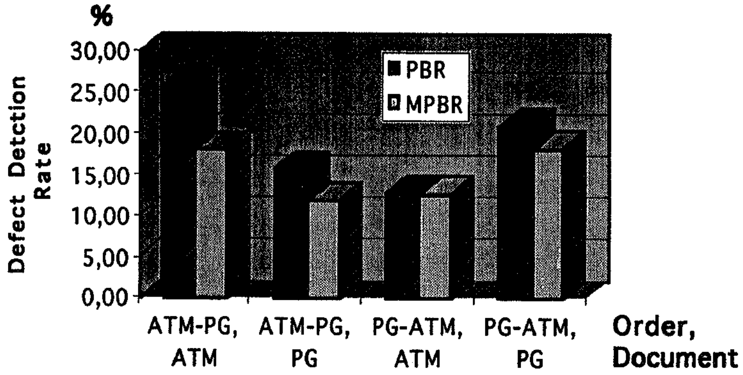
Improved conformance?

Effect on Conformance



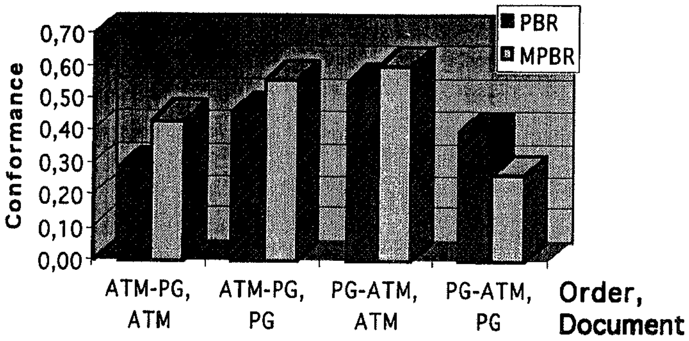
No significant difference.

Effect on Defect Detection Rate



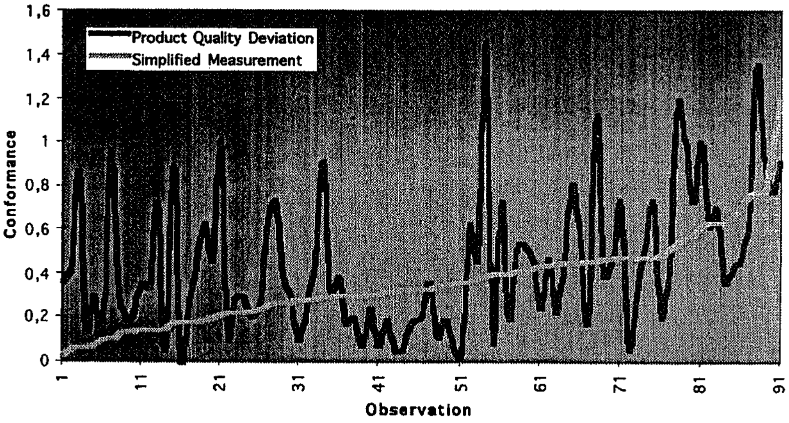
Both significantly worse on second document.

Effect on Deviation in DDR



No significant difference

DDR Deviation vs. Conformance



Significant association.

Conformance in Experiments

Pros

- Reduced variability in process - improved statistical validity.
- Ensure/measure conformance - improved construct validity.

Cons

- Process modifications may be necessary.
- Temporary or permanent modifications?
- Interaction effects with technique type.
- .Conformance at a lower level - where to stop?

Useful when obtaining knowledge?

Conformance in Software Development

Pros

- Reduced variance in product characteristics - better control.
- Improved predictability.
- Ensure valid process-related knowledge.

Cons

- Sensitive data collected.
- Reactive effects - data could be misused.
- Bureaucracy - administrative overhead.
- Reduced performance.

Useful when applying knowledge.

Conclusion

Findings

- Quality deviation and process conformance are correlated.
- No significant effect from modifications.
- Need to test validity further.

Applicability

- Conformance may be useful for
  - Experiments.
  - Process improvement.
  - Situations involving knowledge transfer.

Some benefit, but further investigation needed.

Omit
this
page

Session 3: Using Commercial Software

Using a Unified Object Topology to Uncover COTS Integration Challenges and Assembly Affinities
W. Tepfenhart and J. Cusick, AT&T

Product Development with Massive Components
K. Sullivan, J. Cockrell, S. Zhang, J. Knight, University of Virginia

Technology Evolution: COTS Transition at Raytheon 1983 - 1996
T. Lydon, Raytheon Electronic Systems (RES)

Using a Unified Object Topology To Uncover COTS Integration Challenges and Assembly Affinities

S7-61
415651
360706
p.20

William Tepfenhart
AT&T
Middletown, NJ
william.tepfenhart@att.com

James Cusick
AT&T
Bridgewater, NJ
james.cusick@att.com

Abstract. Large corporations are attempting to cut development costs by relying on integrating several COTS to achieve partial or complete business solutions. That the benefits are not turning out as expected is slowly becoming a recognized issue. In this paper, we address some of the reasons why integration of COTS is a challenge.

Introduction

As corporations move from monolithic single technology systems to large hybrid distributed systems based on multiple technologies, it is becoming increasingly important to understand how these technologies can work together. Often, the desire to integrate two technologies arises from a realization that two very different programs, when combined together, should provide exactly the functionality required for a business need. This recognition usually results in what is termed an integration effort rather than a development effort. Shortened deadlines and reduced funding result - a practice justified by that fact that so much of the functionality already exists. After all, why should we pay money to write a complete solution from scratch when we can purchase two relatively inexpensive products (each of which provides half the functionality) and link them together. How tough can it be? How long can it take? How much can it cost?

It does not take much practical experience before one realizes that a tremendous amount of development effort is required to combine two programs (or program fragments) to meet a specific business need. The projected time and cost savings do not manifest themselves. Answers as to why costs were so high often sound like lame excuses - we had to write adapters to get them to work together, this function didn't work with that function like we expected, and we had to write additional code to meet some of the requirements that weren't met by either product, the purchased product didn't have the specific labels used by our organization and we had to rewrite them, etc. ad nausea.

The feeling that these are lame excuses does not negate one important point. Real work has been performed to make the integration functional. This is a major source of management dissatisfaction with integration efforts. Why should integrating two products require almost as much work as building a system from the ground up?

This paper introduces a technology topology as a tool for understanding COTS integration issues. It explores the issues by demonstrating the dissonance between two technologies and the extra effort required to get them working together.

A Tool For Understanding

In early 1995, we started looking at how diverse Object-Oriented concepts worked together to assist in the development of large-scale systems. In particular, we wanted to know how we could use our knowledge in areas of domain modeling, architectural styles, frameworks, kits, and object-design patterns to ease and stream-line the development of a system. The result of this effort we termed an Object Topology in the

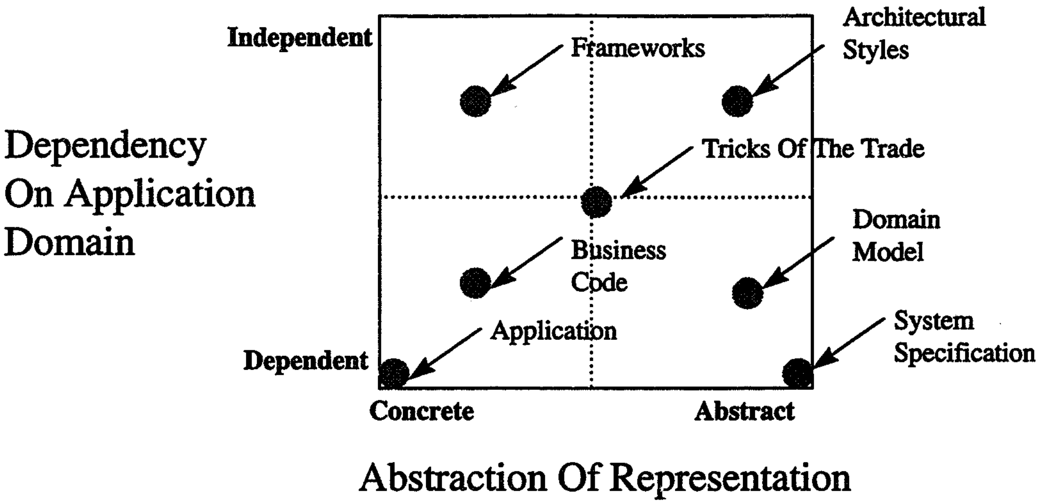


Figure 1. Technology Topology and representative technology components.

sense that it provided a road map of how these technologies worked together. Our result was documented in a previous paper [Tepfenhart].

Since that time, we have extended our investigations to include other development paradigms including conventional procedural programming, relational databases, artificial intelligence, and web technology. In the course of our investigations it became very clear that we were dealing with very diverse technologies, each of which has its own vocabulary and supports software development in very different ways. It turned out that each technology has its' own topology - a different road map for getting from requirements to working system.

Technology Topology

This section of this paper introduces the concept of a technology topology. A topology is a description of the properties of a surface. A road map is one example of a topology. Road maps use longitudes and latitudes as the basis for organizing the points on the map. Longitude and latitude are the coordinates for the topology.

A technology topology is a road map for using a software programming technology. In a technology topology, we use the abstraction of the representation and the application domain dependency as the two coordinates by which we organize points on the map. Elements of a technology that use pictures and/or natural language are said to have a very abstract representation. Elements in which the representation can actually be executed are said to be very concrete. Machine code is very concrete, source code is moderately concrete, a design diagram is moderately abstract, and a requirements specification is very abstract. An element of a technology that is expressed in terms that have little to do with the application domain is said to be application domain independent. Conversely, an element of a technology that is expressed entirely in terms of the application domain is said to be application domain dependent.

We can create a matrix of different elements of programming in terms of their general location in a technology topology. In the paragraphs that follow we will explore where the different elements lie on the generalized topology. Figure 1 shows the completed topology and representative technology components in terms of their approximate placement on the topology. However, different technologies have elements that lie in slightly different points on the topology. This will be shown in the next major section.

A system specification is highly application domain dependent and very abstract. This is because a system specification is typically expressed in natural language and deals with application domain concepts. The system specification for a billing system is necessarily expressed in terms of words associated with billing concepts. A good system specification is essentially technology independent -- one is interested in what the system will do, not if it is implemented using O-O or relational approaches. In practice, system specifications are often expressed in technological terms.

A domain model is highly abstract and very domain dependent. It is highly abstract because graphical representations are often employed. It is moderately domain dependent because the design presents a structure that captures application domain concepts. In most technologies, the domain model forms the basis of a design. Such a design for a billing system would have components named print_bill where bill is a domain dependent term.

An architecture is highly abstract and moderately application domain independent. The components of an architecture are elements which fit specific architectural styles. The representation is abstract since architectures are usually captured in a diagram. Architectures and architectural styles are application domain independent because they deal with things like platforms, files, processes, and protocols. None of these are described in terms of application domain elements.

An implementation of an architectural component is a framework. These elements are typically COTS systems. A framework is moderately to highly concrete and highly application domain independent. For example, a client-server architecture which has PowerBuilder and Sybase components is highly concrete and very domain independent. PowerBuilder and Sybase have no concepts built into them of any particular domain. The power of such COTS architectural components is that they are application domain independent while providing a large degree of functionality. Application concepts have to be added as an additional step in development.

The parts of a program added onto the basic implementation of an architectural component are moderately concrete and very application domain dependent. Application domain concepts identified in a domain model are captured in a programming language. Programming languages are moderately concrete forms of representation (an executable version is easily achieved by compiling and linking). In a programming language, concepts are captured in the form of variable names (PO_Number) and operations (compute_total_bill).

An executable program is highly concrete and very application domain dependent. That is, an executable program runs and thereby provides the functionality described in the a system specification. Executable programs are necessarily technology independent. That is, we can't tell by it's executable code if it was implemented using O-O, relational, or AI technologies.

There is a final technology component that has only recently become recognized. This component captures the 'Tricks Of The Trade'. These are the programming heuristics, rules, and patterns that describe how to recapture one technology component into the representation of a second technology component. Programming practices are generally neutral in application domain specificity and neutral in abstraction. On one hand, they describe patterns of domain terms. On the other, the patterns are usually in terms of very abstract domain terms. They relate highly abstract representations with rather concrete implementations of the information.

Development Geodesic

If we examine a technology topology, one sees that there are islands of technology components that are reflected across the line of neutral abstraction. A system specification is reflected by an executable program. A domain model is reflected by an implementation of the application domain component. An architecture is reflected by a framework. The mapping of an abstract representation to a concrete representation is a development activity.

A development geodesic can be viewed as the path of least resistance in the development of a product from a point on the abstract side of the graph to its counterpart on the other side of the topology. Three such geodesics exist on the topology that represent the reflections across the line of neutral abstraction. These are shown in Figure 2.

The dashed path describes the development route for taking a domain model into a set of business code. The path traverses through the 'tricks of the trade' node - a practice which real developers perform to

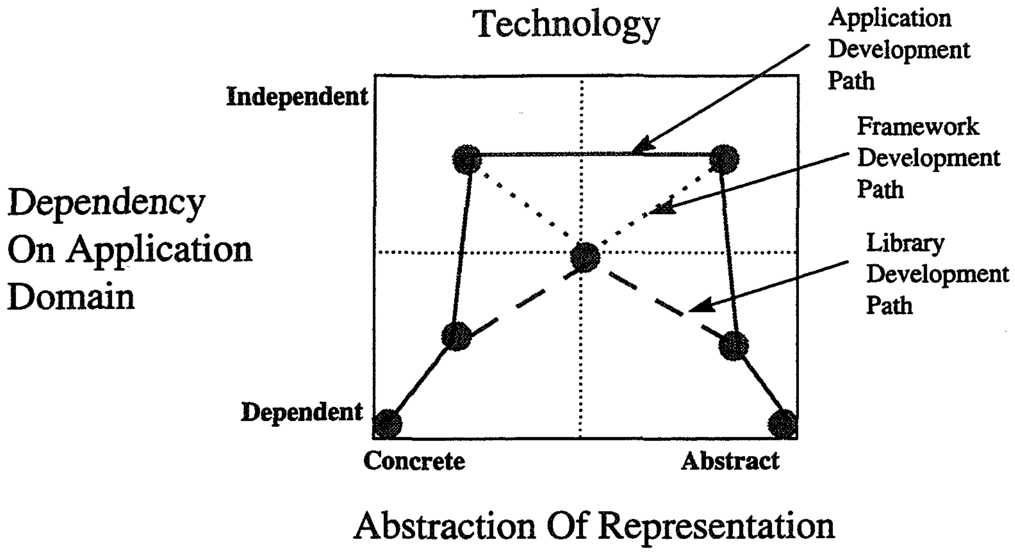

Technology
Application Development Path
Framework Development Path
Library Development Path
Independent
Dependent
Dependency On Application Domain
Concrete
Abstract
Abstraction Of Representation

Figure 2. Development Geodesics.

achieve high quality and high performance code. In the object-oriented development world, these tricks of the trade are object design patterns which tell how to map an object model into an object implementation.

The dotted path describes the development route for taking an architectural style into a framework. Again, this path traverses through the 'tricks of the trade' node. In this case, some of the tricks of the trade are the identification of COTS products that provide a basic framework for an application. These include products like X-Windows, DBMS, Web Servers, and others.

The solid path describes the development route for taking a system specification into an operational application. While not really drawn, this path takes into account the other two development paths as well. The core path travels from system specification to a domain model and then onto an architecture. This is all work performed using abstraction representations. From those points, development is being performed to map them onto frameworks, sets of business code and then integrating them into an application. The development effort required to translate the abstract representations into concrete representations for domain models and architectures will follow the paths described previously.

The lines between nodes represents a kind of effort to tie all these technologies together. The traversal from system specification to a domain model is traditionally a design process. A common manifestation of the design process is a requirements traceability matrix. A top-level design is mapped onto an architecture.

Diverse Technologies

There are many software technologies that have reached maturity. With maturity, we now try to exploit the strengths of each in obtaining critical business solutions. In the following sections, we examine several technologies and identify the topologies. In all cases, the axes remain the same -- abstraction of the representation and dependency on the application domain. The difference among these technologies are the specifics of the locations on the topology of the components and the geodesics connecting the points. This will become obvious as we describe the topology of each technology. In particular, we will see that they differ even in terms of the words used to express basic programming concepts.

Object Topology

Object Oriented approaches to solving business problems have resulted in a number of very large, reliable, and functional systems. They are becoming the cornerstone of businesses as they demonstrate the ability to

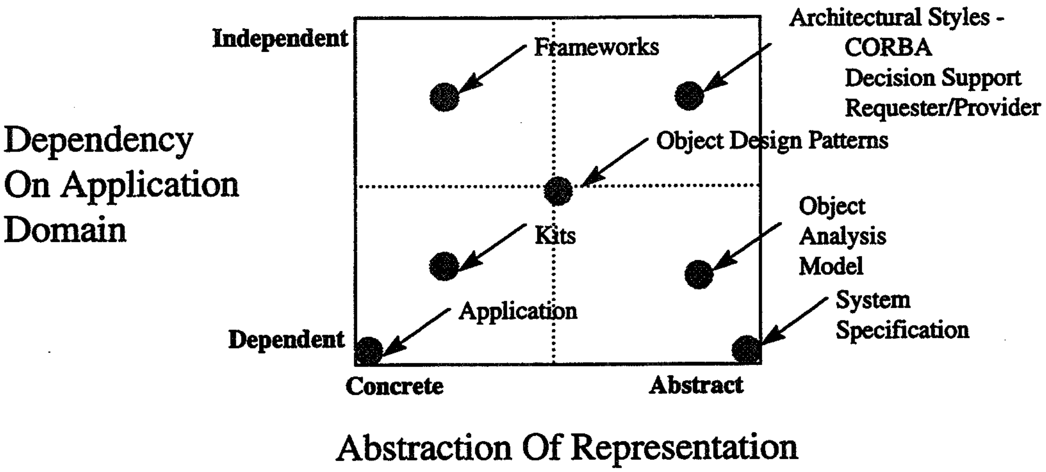


Figure 3. Object Topology.

rapidly adjust to changing business requirements. The topology for the object-oriented paradigm is shown in Figure 3.

In an object topology, a domain model is expressed in terms of an object analysis model. In this technology, domain elements are captured in the form of objects, relationships among objects, and behaviors which objects can exhibit. Class systems for objects are communicated in the form of graphic illustrations with relationships expressed as links or attributes. Behaviors are captured in the form of event-trace diagrams.

Certain architectural styles are common in object systems. In particular, one deals with architectural styles such as decision support, requester/provider, and event-driven styles. These are reflected in the types of frameworks and COTS available for object systems. In particular, one has MFC from Microsoft, Zapp from Rougewave, ObjectStore from Object Design, and Orbix from Iona to name a few.

One area in which there has been a lot of major research of late concerns the 'tricks of the trade' technology component for the object paradigm. This has lead to a clear set of specifications concerning how to map the object model into source code. These specifications are object design patterns and the use of these patterns is becoming increasingly more wide spread.

Relational Topology

The technologies associated with relational data bases maps into a topology of its own. Relational systems have long held a major role in business applications. The topology for the relational paradigm is shown in Figure 4. This topology should be compared with the one for the object paradigm.

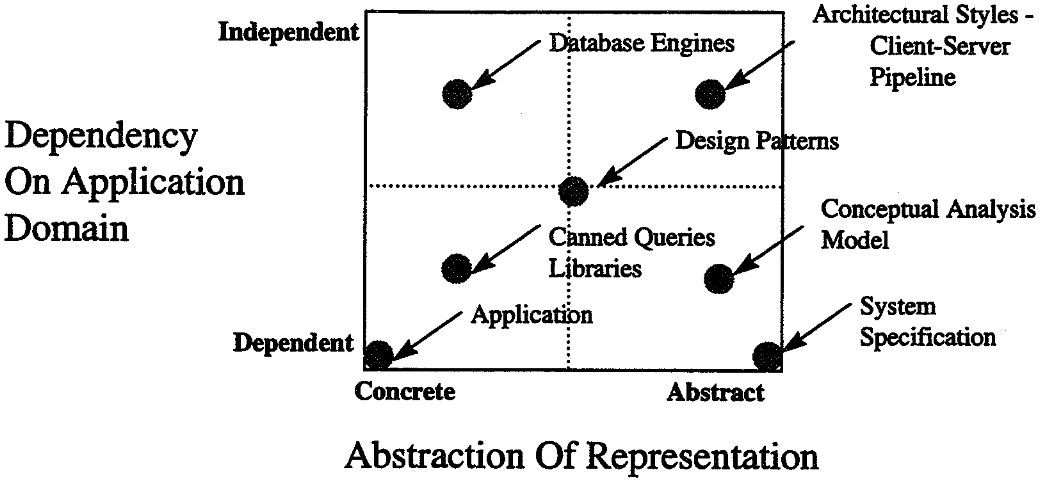


Figure 4. Relational Topology.

In a relational topology, a domain model is expressed in terms of an entity-relation model. In this technology, domain elements are captured in the form of tables, relations among tables, and operations over table entries. The entity-relation model is expressed in the form of graph illustrations that reflect a table view of the world. An entity-relation model is very different from an object model.

Certain architectural styles are common among relational systems. The most widely known is the client-server architecture in which their is a common server and any number of clients that may be presentation systems and/or decision support systems. These architectural elements are reflected in the COTS products available for relational systems. DBMSs are one kind of product available on the server side and client-side products like PowerBuilder are becoming more widely used.

The 'tricks of the trade' technology component are being captured in the form of Design Patterns which relate how different domain models can be implemented in tables and queries over those tables.

AI Topology

AI is often an overlooked technology in obtaining business solutions. However, rule based systems are still quite a factor in the software enterprise. The AI topology is shown in Figure 5. In a vein appropriate to the fuzzy heuristic driven AI world, developers often talk in terms of development heuristics instead of design patterns.

In an AI topology, a domain model is expressed in terms of a knowledge level analysis model. The principle concepts involved are: facts, predicates, rules, and chains of inference. These elements are usually captured in natural language form.

AI systems are usually implemented as either consultation systems or embedded systems. The COTS products are limited to inference engines and development tools that support either mode of operation.

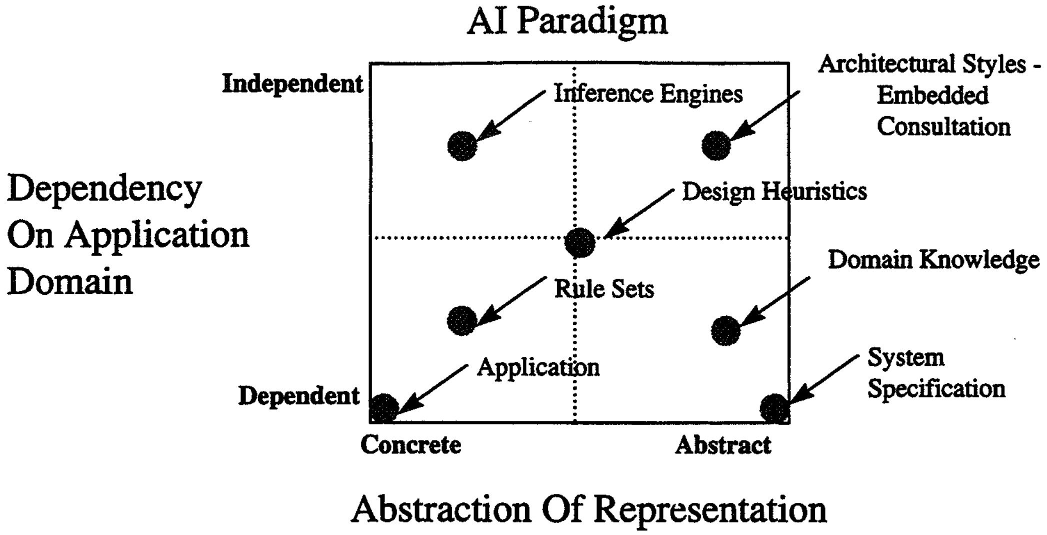


Figure 5. AI Topology.

Web Topology

One of the hottest technologies in the market place today is web technology. This promises to solve many of the problems associated with large scale use of applications in non-homogeneous computing environments. The browser, available across many platforms, provides a front-end to an application on a back-end machine. The topology for web technology is illustrated in Figure 6.

In a web topology, a domain model is often expressed in terms of pages, forms, and state models. In this technology, information is presented as a page of material, a form to be filled out, or as a single snap-shot in a sequences of pages. Expression of design is achieved using CGI bin scripts and HTML documents.

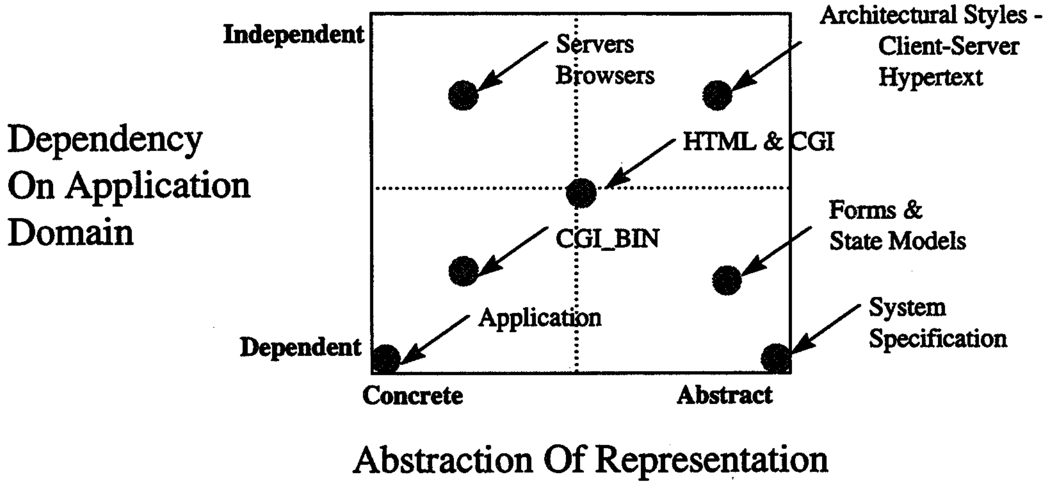


Figure 6. Web Topology.

In terms of architecture there are client-server systems and hypertext documents. COTS products include the servers, the clients, and some authoring tools. There are some COTS systems that provide a page generation between the web server and permanent data stores (such as DMBSs).

The development geodesic is currently poorly understood since individuals are still exploring the topology and the topology is still undergoing tremendous changes. One of the most common appears to layout the basic page appearance and to implement whatever processing needs to be performed as a single cgi-bin program on a page-by-page basis.

Building Hybrid Systems

It is clear that as systems get larger and more complex that the strengths of any one technical approach will fail to meet business needs totally. To counter this, mixtures of technical approaches are being employed.

If we were to place any two technology topologies one atop the other, we would see that each has the same components, but the components are placed in slightly different locations. As suggested in the previous section, this is because the different technologies provide different abstractions for expressing application domain concepts. The result of mixing two paradigms and trying to treat them as a single technology is illustrated in Figure 7.

A key to understanding the problems associated with mixing technologies to recognize is that two points now reside in each area where a single point used to exist. There are now two different kinds of domain models, one which is appropriate for one technology and another for the other technology. There are two points for architectural styles, each point identifying a set of architectural styles appropriate for the individual technologies. There are two points in the frameworks region denoting that there are different architectural styles being implemented for the two different technologies (and the fact that there are different COTS products). Finally, there are two sets of business code reflecting the fact that two different domain models are being captured.

The significance of this picture becomes most apparent as a result of tracing the development geodesics on the topology. The development paths become much more complicated since we have the existing paths for each technology and additional segments that have to be added to connect the dual points. It is necessary for the connections between dual points to be made so that one ends up with a fully functional application. In particular, if the two sets of business code do not integrate seamlessly, then the application won't function. In order for the two sets of business code to integrate seamlessly, then some sort of integration models must exists for the two domain models.

If one takes the superficial view that each segment of a development geodesic represents some standard unit of work that must be performed during development, it is obvious that more work is required to implement an application. In fact, one could easily be convinced that mixing technologies can require almost three times as much work as implementing from scratch within a single technology. The necessary work could be computed as the sum of the work required for one technology plus the work for the second technology plus the work for integrating them.

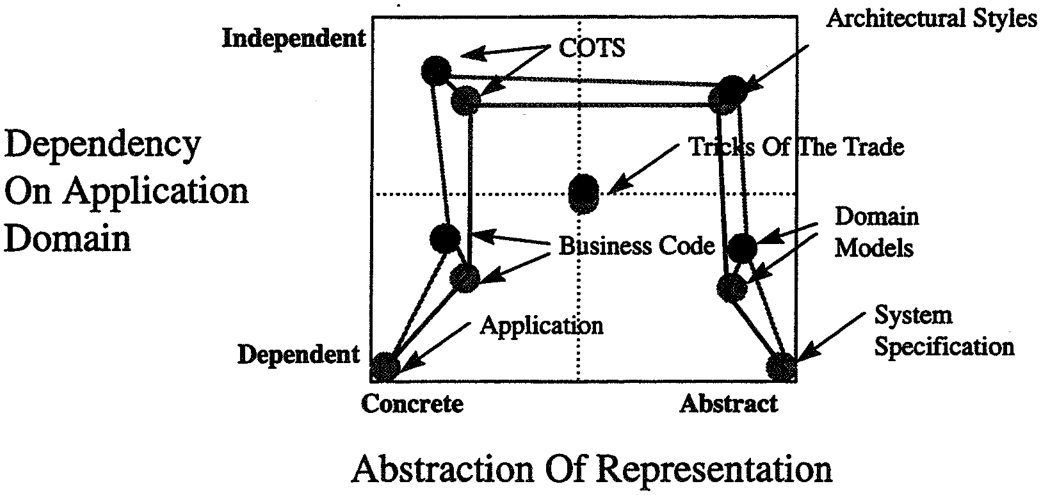


Figure 7. Mixing Two Different Paradigms.

Of course, considering each link to represent some standard unit of work is a superficial view. This view ignores some of the advantages which one has when COTS products are employed in a system. However, the introduction of a COTS product does not eliminate the work entirely. Hence each link does represent some amount of effort, but that amount will differ according to technology and COTS products supporting it. That is why using a COTS product in an application can be cost effective.

There is another observation that can be made on the basis of Figure 7. This observation is that while the individual paths for each technology can be well known and understood, the little development links necessary to connect the development nodes can be virtually unknown. This is shown by the fact that there aren't any 'tricks of the trade' for linking two technologies.

In essence, one aspect of using two technologies is identifying how the domain models can be linked together, what architectural styles work well together, how to connect frameworks, and how the business code can be integrated across technological abstractions. This kind of unique, first of a kind activity is one that can require time and money. Further more, it carries with it a high degree of risk.

The trade-off concerning the relative costs of staying within a single technology or mixing technologies has to be made on a total cost perspective. Staying within a single technology might lead to high costs because of the effort associated with developing of a major functionality which is not provided in any other way. On the other hand, the cost of mixing two technologies may be high because of the effort of development for each technology and the difficulties in connecting them together.

Summary

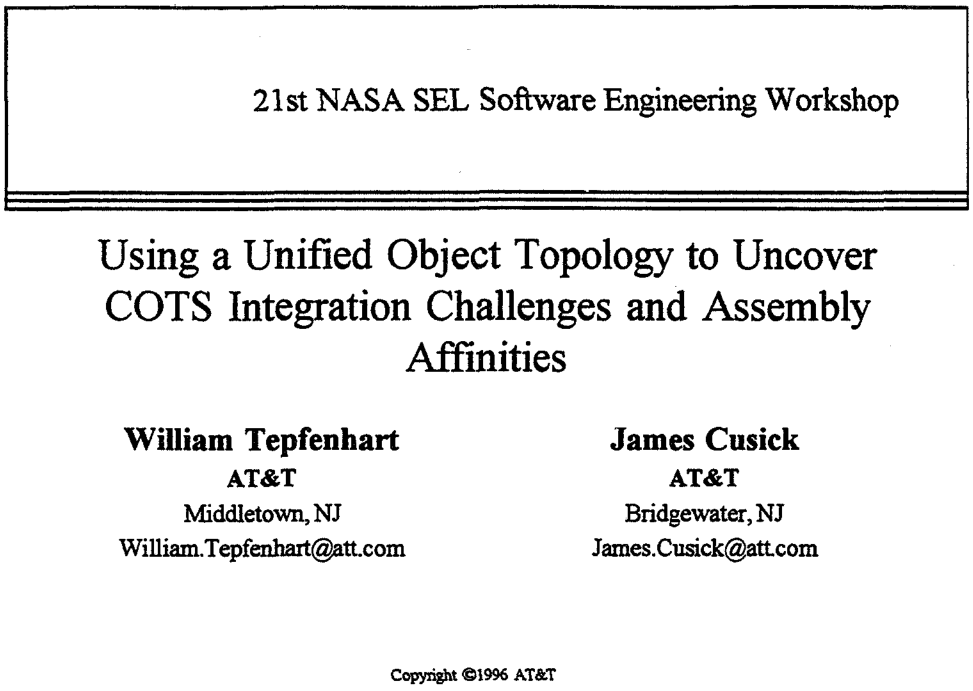
This paper has presented a tool, the technology topology, for understanding COTS integration challenges. It used this tool to describe the relative relationships among the components of a technology. It described how development of an application traverses a geodesic across the topology.

A major section of the paper dealt with the different kinds of software technologies. It identified the basic concepts and laid them out on the topology. This was done as a preparation for demonstrating how two different technologies fail to overlap on the topology. The naturally arising differences in location on the topology was identified as the major source of integration challenges. It showed how development of a system using two separate technologies could easily require much more work than development from scratch using a single technology.

We did not cover some issues associated with COTS integration. These issues include a lack of maturity of a technology. An example of this is the Web Technology which is still in its' infancy and one in which not all of the major technological components have been adequately developed. Another issue deals a lack of maturity in a product. Not all products are equally mature within a technology. This lack of product maturity can demonstrate itself as a lack of basic features or inconsistant application of a technology. Also, an immature product can be very buggy. Each of these issues raise additional integration challenges as developers try to work around bugs in one product by implementing a feature in another.

References

[Tepfenhart] Tepfenhart, W.M, and J. Cusick, "A Unified Object Topology," IEEE Software, January, 1997.



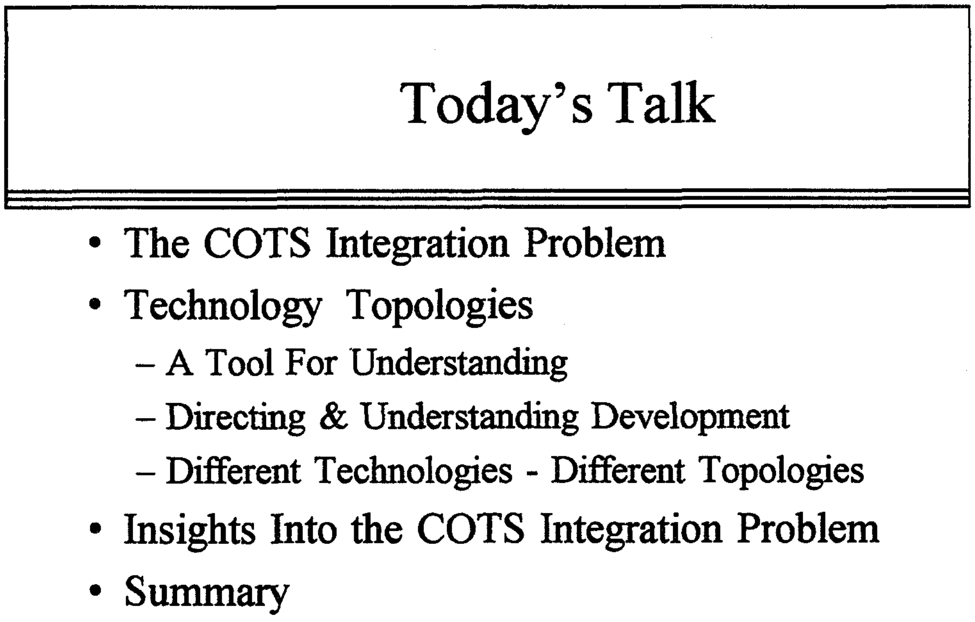
21st NASA SEL Software Engineering Workshop

Using a Unified Object Topology to Uncover COTS Integration Challenges and Assembly Affinities

William Tepfenhart
AT&T
Middletown, NJ
William.Tepfenhart@att.com

James Cusick
AT&T
Bridgewater, NJ
James.Cusick@att.com

Copyright ©1996 AT&T



Today's Talk

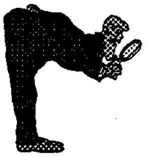
- The COTS Integration Problem
- Technology Topologies
  - A Tool For Understanding
  - Directing & Understanding Development
  - Different Technologies - Different Topologies
- Insights Into the COTS Integration Problem
- Summary

COTS Integration Problem

- The COTS Integration Problem Arises Whenever Someone Observes That Their Business Need Can Be Satisfied By Two COTS Products
- All They Have To Do Is *Just* Integrate Them Together

Just Integrate Them Together?

- A Few Quick Questions:
  - How tough can it be?
  - How much can it cost?
  - How long can it take?



- A Few Hard Won Answers:
  - It can be very tough
  - It can cost a LOT
  - It can take a long time

Reasons Sound Like Excuses

- One Product Assumed Certain Things Inconsistent With The Other Product.
- We Had To Write A Lot Of Code To Get Them To Talk To Each Other.
- We Had To Modify Them To Talk With Our Other Systems.
- The Presentation Of Information Didn't Really Follow Our Corporate Standards.

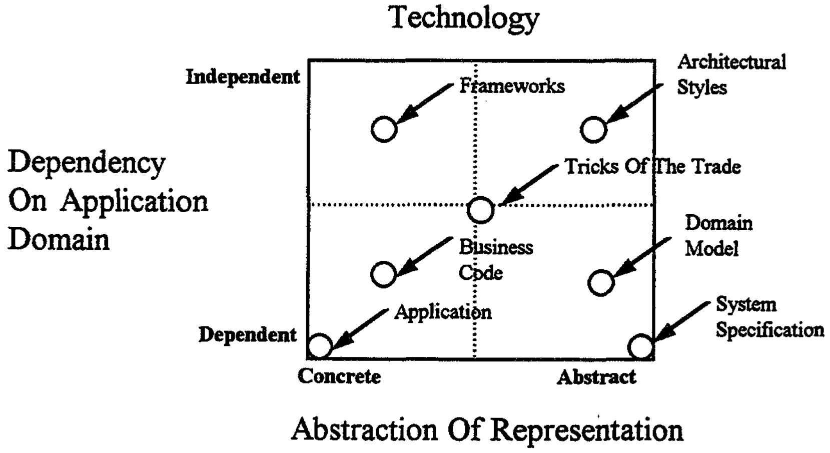
In Failure We Forget

- Real Effort Was Expended To Get It To Work
- There Are Real Reasons Why It Is Tough To Integrate COTS software!

A Tool For Understanding Why

- Technology Topology
  - A Roadmap Relating Different Technology Components
  - Development Methods Are Geodesics For Traversing The Topology
  - Clarifies Integration Problems

Technology Topology



Technology

Independent

Dependency
On Application
Domain

Dependent

Frameworks

Architectural
Styles

Tricks Of The Trade

Domain
Model

Business
Code

Application

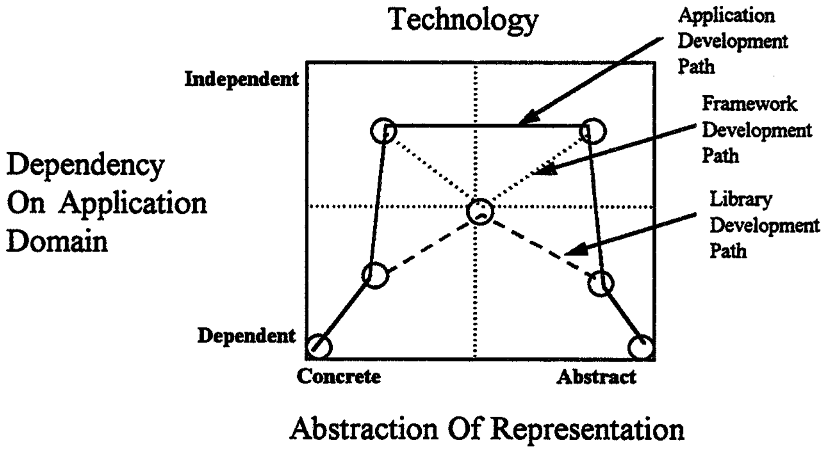
System
Specification

Concrete

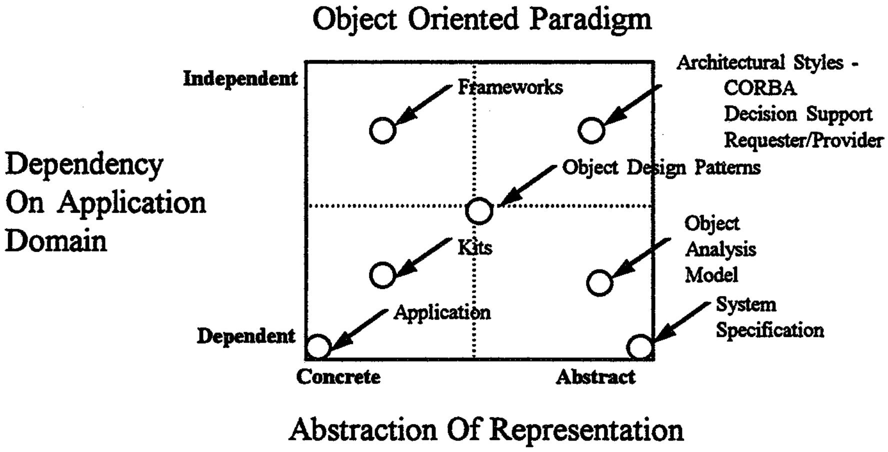
Abstract

Abstraction Of Representation

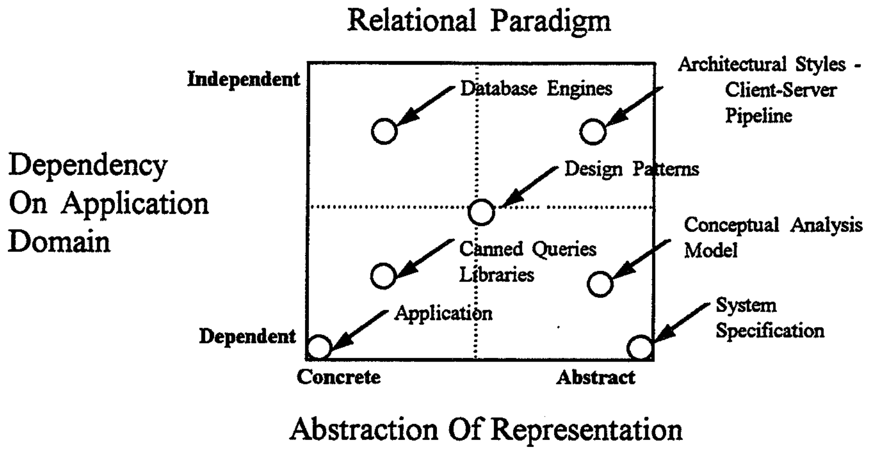
Development Geodesics



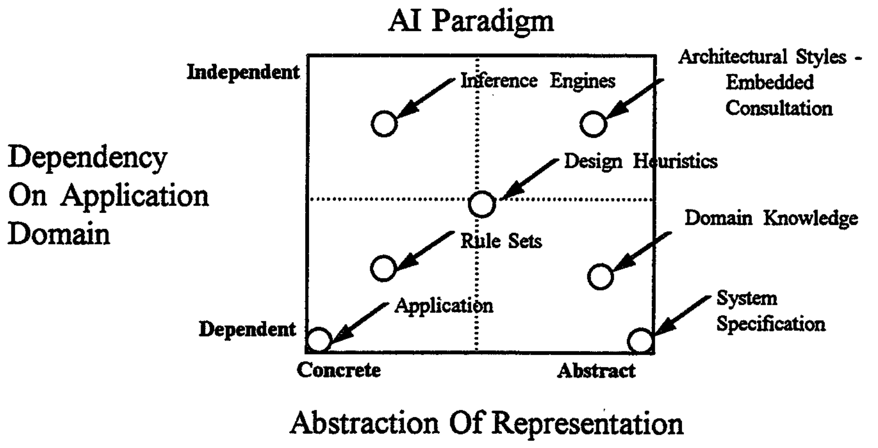
Object Topology



Relational Topology

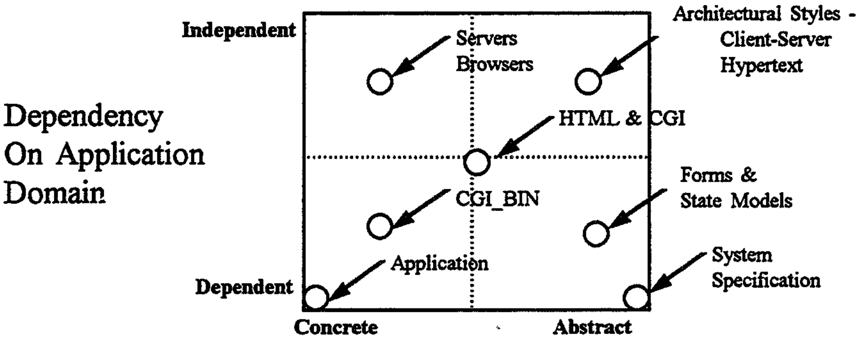


AI Topology



Web Topology

Web Paradigm



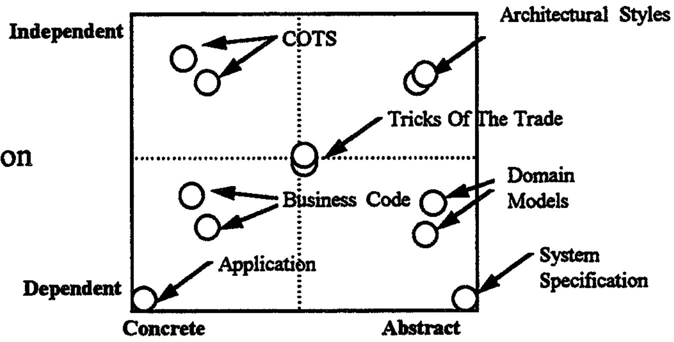
Abstraction Of Representation

General Comments

- System Requirements Are Independent of Technology
- An Application Is Just An Application
- Different Architectural Styles Support Different Technologies
- Some Points Of Topology Are In Very Different Locations

Hybrid Topology

Two Different Paradigms


Independent

Dependent

COTS

Architectural Styles

Tricks Of The Trade

Business Code

Domain Models

Application

System Specification

Concrete

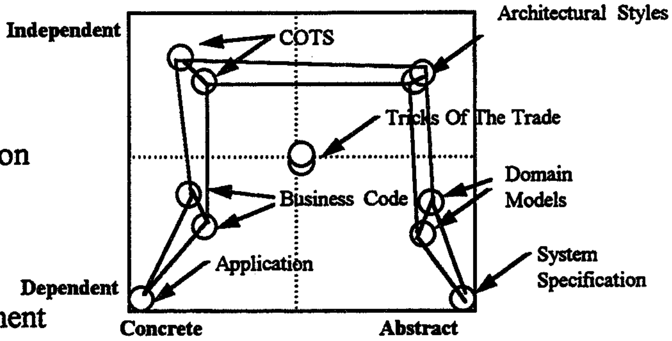
Abstract

Dependency On Application Domain

Abstraction Of Representation

Application Development

Two Different Paradigms


Independent

Dependent

COTS

Architectural Styles

Tricks Of The Trade

Business Code

Domain Models

Application

System Specification

Concrete

Abstract

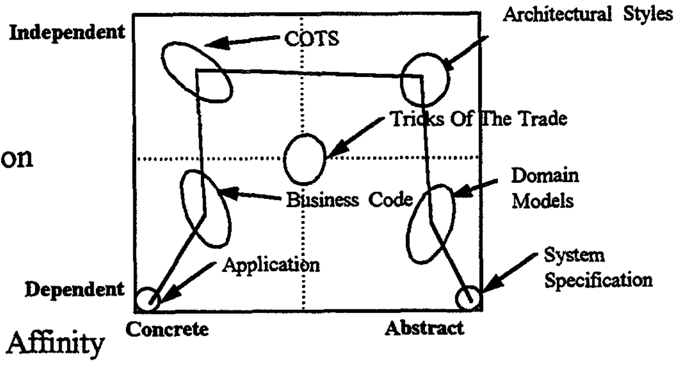
Dependency On Application Domain

Two Development Paths and Connections Between Them

Abstraction Of Representation

Is It A New Paradigm?

Union Of The Two Technologies Plus a Little Bit From Their Integration



Dependency
On Application
Domain

An Integration Affinity
If Vendor Provided!

Abstraction Of Representation
Entirely New Development Path

Factors Not Addressed

- Maturity Of Technology
  - No Development Path Defined
  - Missing Technology Components
- COTS Product Maturity
  - Bugs
  - Inconsistencies
  - Technological Disconnects

Summary

- COTS Integration Issues Arise Naturally
- They Are Complex In The Sense That They Are Present At Each Step In Development
- Vendors Can Help By Providing Hybrid Development Paths

PRODUCT DEVELOPMENT WITH MASSIVE COMPONENTS

Kevin J. Sullivan, John C. Knight, Jake Cockrell, and Shengtong Zhang

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

{sullivan | knight | emc5a | sz2n}@virginia.edu
(804) 982-2206

58-61
415652
360708
P20

An Abstract Submitted To:

The Twenty-First Software Engineering Workshop
Goddard Space Flight Center
Greenbelt, MD 20771

Introduction

The problem faced by many of today's software engineers is to build and maintain broad families of large systems in a cost-effective and *timely* manner. Because the market demands rapid creation and modification of systems in response to a spectrum of evolving requirements, extensive flexibility in systems is required. This situation has two implications: first, basic system demands have to be met quickly; and, second, responses to requested variations have to be rapid and effective. System development and modification cycle time must be shortened significantly.

One approach to cycle-time improvement that has been studied extensively is software reuse. Current reuse techniques include system synthesis using application-generator technologies and component-based development techniques. The latter has been effected in several ways, including subroutine libraries, templates, and a variety of class and framework mechanisms.

On the basis of some experimental systems work, we suggest that a relatively new approach might merit increased attention from the research community. The approach is based on the integration of large, application-scale, binary components. To date the approach has been employed industrially using shrink-wrapped packages, such as Microsoft Office and Visio Corporation's Visio technical drawing tool, mostly for business and office automation tasks.

We have shown that this approach can be applied more aggressively, using today's technology combined with advanced integration strategies such as *mediators* [S94], to develop systems in at least one domain far removed from business data processing, quickly and at low cost. Our demonstration application is a fault-tree analysis tool embodying new analysis techniques developed by Joanne Bechta Dugan at the University of Virginia.

We cannot infer broad generality from a single example. However, it does appear that our approach can be applied in developing a range of modeling and analysis tools using existing technology. The approach does appear to overcome some previously encountered impediments to large-scale reuse [G95]. That, however, is not the main point of this abstract. More importantly, our success applying the approach in an engineering domain suggests the hypothesis that we can thoroughly characterize, develop, and generalize it, so as to enable its application to solve problems in a significantly wider variety of problem domains.

A key problem is that we do not yet understand very well what features of the approach account for its success even in the limited domains of business data processing and tools. We have decided to focus part of our research on answering this question. A first objective is to determine what general features of the approach account for successes to date. A second objective is to determine what is required to develop and generalize the approach, so as to apply it more aggressively and systematically to problems in domains in which the existing technology is inadequate.

In this abstract, we present early answers emerging from our attempts to determine which general design properties account for the success of this large-scale integration approach. We begin with an analysis that suggests why it seems to offer perhaps greater promise than previous, smaller-scale reuse approaches.

Component Size

It can be argued that reuse in which engineers attempt to develop systems by reusing small building blocks does not attack the essence of the problem. Consider, for example, a system that ends up being one million lines long. Even if the *entire* system is build with C++ reusable classes and the classes are say 100 lines long on average, the total number of items being composed in 10,000. As well as understanding and using the classes themselves, the system developers have to design and implement the interconnections among the components, and maintain intellectual control over these interconnections. That is a massive design task that would appear to be inconsistent with fast cycle time and low cost. Developing a system with such building blocks remains a tremendous challenge, and the total software engineering burden has been reduced some but not enough by reuse of small building blocks.

Achieving truly significant benefits from component-based reuse would appear to require the reuse of massive components so as to enable large systems (for example, one million lines) to be constructed by straightforward integration of just a few components. With this goal in mind, it is clear that components that average 100 lines in length are too small by about three orders of magnitude. Despite the obvious benefits, attempting to reuse large components has met with only limited success. One problem, as reported by Garlan et al. [G95], significant difficulties can arise with what has been referred to as *architectural mismatch*; but this is by no means the only problem.

An even more aggressive view is that successful development based on the reuse of massive components is unlikely to be realized by incremental improvements in the size of typical reusable components from their present small size. Is it necessary for larger component sizes to come about only incrementally as more is learned about building flexible components? The experiment that we are conducting has shown the feasibility of using massive components today, and suggests that an immediate transition to the use of massive components is possible, at least in certain cases. As an alternative to trying to make progress by "climbing up" from the use of small components, we suggest starting with a massive component approach and "working backwards" as difficulties are encountered.

Our use of massive components is different from the way in which components are used in a traditional systematic reuse approach. The components that we are using each provide tremendous functionality, and each is many hundreds of thousands (possibly millions) of source lines in length. Despite this, we have found the integration of these massive components to be successful in the senses that they were easy to use and the resulting product performs as required. In view of their size and functionality, we think that it is important to distinguish between the more traditional notion of component and the type of massive component that we are using. We have coined the term "application service" to describe the latter and will refer to such components using this term throughout the remainder of this presentation.

USING APPLICATION SERVICES

In an earlier paper [SK95], we reported preliminary results of an experiment on large-scale systematic reuse. We described our experience with efforts to exploit an architecture (Microsoft's OLE) that permits very large components to be integrated. We used this architecture enhanced with mediators [S94] and several application services to develop a high-quality, industrial-strength software toolset. Our conclusions were that the basic architectural concept worked well, although several technical difficulties remained.

The high-level architecture of the toolset that is the subject of the experiment that we are conducting is shown in Fig. 1. The toolset provides facilities for a technique called system fault-tree

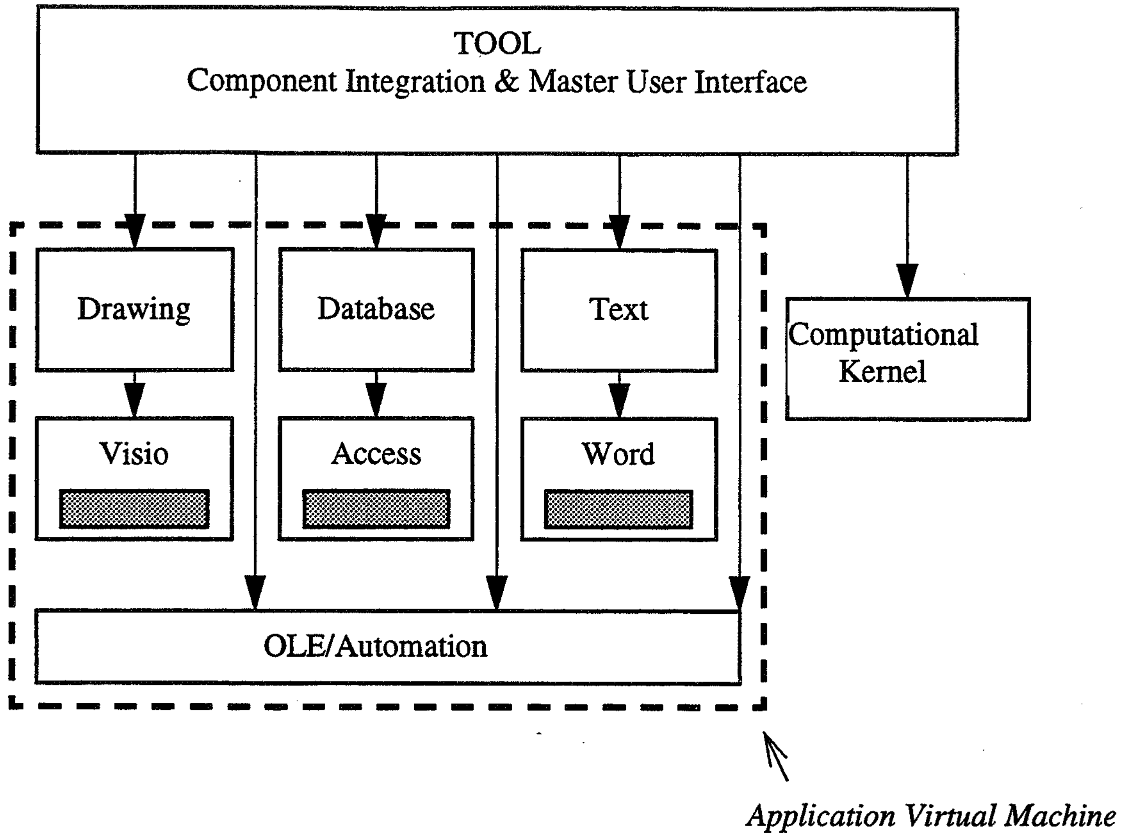

TOOL
Component Integration & Master User Interface
Drawing
Database
Text
Computational Kernel
Visio
Access
Word
OLE/Automation
Application Virtual Machine

Figure 1 - Toolset architecture.

analysis—a technique used in reliability engineering [V81]. A main program is responsible for providing the user's primary control mechanism, and also initiates execution of the required application services. Three application services are used: Visio Corporation's Visio technical drawing program, Microsoft's Access database program, and Microsoft's Word text processing program. Visio is used to provide a graphic representation using customized icons of the fault tree of interest together with a graphic (click and drag) editing facility. Access provides a general database facility that is used for storing fault trees and various forms of failure data used in the analysis. Word is used to edit an ASCII representation of fault trees that is useful for certain kinds of fault-tree creation and editing. These three application services are supplemented with mediators that provide links between the application service and certain canonical data structures maintained by the main program. Critical reliability analysis functions are available to the main program in a conventional form as a set of classes (shown in Fig.1 as the computation kernel).

The three application services form what we refer to as an *application virtual machine*. It is this virtual machine that the main program manipulates, along with the computational kernel, to provide the toolset's functionality. This manipulation uses subprogram calls as might be used in a traditional design together with action invocation via events.

The toolset that we built demonstrates industrial strength functionality and performance.

Some cosmetic elements of the individual application services remain. The services do provide some support for customizing or removing application-specific interface elements. We intend to remove those that are not needed, to the extent that this improves the toolset's coherence and appearance, and to the extent supported by the existing application packages. Certain key functional elements (such as editing commands) will be left available through the application services' own user interfaces so that they maintain the look and feel of similar products. How best to support integration at the application service user interface level remains a technical—and perhaps a research—issue that has not yet been fully resolved.

Reuse Analysis

In terms of reuse, the results we achieved were successful—we believe significantly more successful than might be expected since the toolset was built using application services. Why is this the case? In this section we present the results of a preliminary analysis of this success.

The application services provide enormous functionality including large and important parts of the basic functionality of the domain. Not only is this functionality provided but it is provided with sufficient flexibility that it can be tailored easily to the specific needs of an individual product. We refer to this functionality as the *critical superstructure* of products in the domain. It is this aspect of many products that consumes the vast majority of the resources yet is not what provides the unique capabilities of the product. In the case of the toolset we have used in our evaluation experiment, for example, many parts of the toolset are commonly found in software tools.

That the flexibility offered by the application services was not overwhelming is counterintuitive. Many efforts to generalize components to meet a variety of needs has resulted in components that are unwieldy. Support for advanced specialization mechanisms seems to play a key role in this regard. Although it is possible to specialize Visio by writing custom code in C++, it is more common to exploit its spreadsheet mechanisms to "program" the behaviors of user-defined shapes. Similarly, one specializes Word by defining new document templates. These mechanisms provide high-level support for flexibility in the dimensions that are actually critical in practice.

Even with the provision of powerful flexibility features, application services cannot be used effectively unless they can be integrated smoothly. The architectural approach used (OLE, mediators, and the application virtual machine structure) allowed a set of application services and product-specific software elements to be integrated so that the resulting system presents a comprehensive unified interface and behavior to the user. This is a significant result since integration involves a variety of invocation and data interchange requirements.

As well as the combination of functionality, flexibility, and integration facilities, a number of other complex aspects of both the application services and the integration mechanism contributed to the successful reuse that we observed. We summarize briefly the main reasons for the reuse success here—more details together with examples will be given in the presentation:

- *Architectural coherence.*
  All of the application services used were designed to work in the OLE environment. This permitted their use in a systematic way and avoided several instances of architectural mismatch.

- *Supplementary use of mediator architecture.*
  We avoided many difficulties by supplementing the OLE architecture with use of mediators in

the toolset design.

- *Provision of the critical superstructure.*
  The application services enabled the creation of the critical superstructure relatively easily.

- *Provision of essential flexibility.*
  The application services provided flexibility in ways well suited to their use in a reuse context.

- *Advanced support for exploiting flexibility.*
  The application services include powerful mechanisms to permit exploitation of their inherent flexibility.

- *Managed object model.*
  The application services provide a managed object model in that they implement an internal object structure that is powerful yet accessible from their application-programming interfaces thus permitting fine-grained integration.

- *Provision for add-on functionality.*
  The functionality of an application service is easily supplemented by creating the requisite additional functionality as a software entity that can be invoked by the user in a number of different (and powerful) ways via the application service.

CONCLUSION

Our conclusion is that by careful design of both application services and the architecture with which they are integrated, large systems can be built successfully using components up to three orders of magnitude larger than components found in typical reuse libraries. We know of no comparable results demonstrating the degree of integration we have achieved at this scale. Our result has not been proven generally, but it has been demonstrated. The demonstration is sufficiently successful, and the reasons why understood well enough, that increased attention to the technical and research issues that have to be resolved to generalize the approach appears to be warranted.

REFERENCES

[G95} Garlan, D., R. Allen and J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard," *IEEE Software*, vol. 12 No. 6, Nov. 1995, pp. 17-26.

[SK95] Sullivan, K.J., and J.C. Knight, "Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse", *Proceedings ICSE 18: Eighteenth International Conference on Software Engineering*, Berlin, Germany (March 1996).

[S94] Sullivan, K.J., *Mediators: Easing the Design and Evolution of Integrated Systems*, Ph.D. Dissertation, University of Washington Department of Computer Science and Engineering, Seattle, WA, 1994. Also available as University of Washington Department of Computer Science and Engineering Technical Report 94-08-01.

[V81] Veseley, W.E., F.F. Goldberg, N.H. Roberts, and D.F. Haasl, Fault Tree Handbook, U.S. Nuclear Regulatory Commission, NUREG-0492, Washington DC (1981).

Product Development with Massive Components

Kevin Sullivan
John C. Knight
Jake Cockrell
Shengtong Zhang

University of Virginia Computer Science

Goal: Radical Improvement

- Productivity
- Quality
- Cycle Time

Means

- Novel Architectural Styles
- For Leading Edge, Real Systems
- Respectful of Key Design Realities
- Explored & Demonstrated by Case Studies

Problem Domain: Tools

- Even Simple Techniques Demand ... (10^4)
- *Massive Superstructures* (10^6)
  - graphical user interface
  - technical drawing
  - text formatting
  - data management
- From-Scratch Construction Uneconomical

Case Study

- Given New Modeling & Analysis Techniques
- Develop Industrial Strength Software Tools
- At Radically Reduced Cost & Cycle Time
- *Dugan's Hybrid Fault-Tree Analysis Method*

Traditional Reuse Inadequate

- E.g., Object-Oriented/Libraries
- 1 Million Lines of Code (10^6)
- 100 Line Reusable Components (10^2)
- Need 10,000 Components (10^4)
- Still A Horrendous Design Problem
- **Doesn't Attack Essence of Problem**
- **Result--Many Terribly Inadequate Tools**

Attacking the Essence

- Simple and Straightforward Integration
- Of a Few Parts (10^1)
- Tailored Quickly and at Low Cost

Observation

- Powerful New Applications
  - Microsoft
  - Visio Corp.
  - Others
- Specializable
- Integratable
- Key Subdomains

Concept

- Application Packages as Components
- Application Virtual Machines
- *Package-Oriented Programming (POP!)*

An Old Idea

"Perhaps the simplest instance of reusability (and the one with the highest leverage) is the purchase of an existing software package. The purchasing organization pays very little compared to building an equivalent capability in-house and it is up and running in a short time. Even if a limited amount of customization is necessary, this is often small compared to the cost of building and entirely new system. If organizations will come to the point of accepting such prepackaged systems, then a major step forward will have been achieved [Horowitz & Munson, IEEE TSE, 1984] ."

Still A Good Idea

"An especially promising trend is the use of mass-market packages as the platforms on which richer and more customized products are built [Brooks, MMM, 1995]"

Questions of Feasibility

- "The [programmer] who uses ... applications as components ... is the user whose needs are poorly met today [Brooks, MMM, 1995]."
- Architectural Mismatch [Garlan 95]
- Lack of Demonstrated Success

Hypothesis

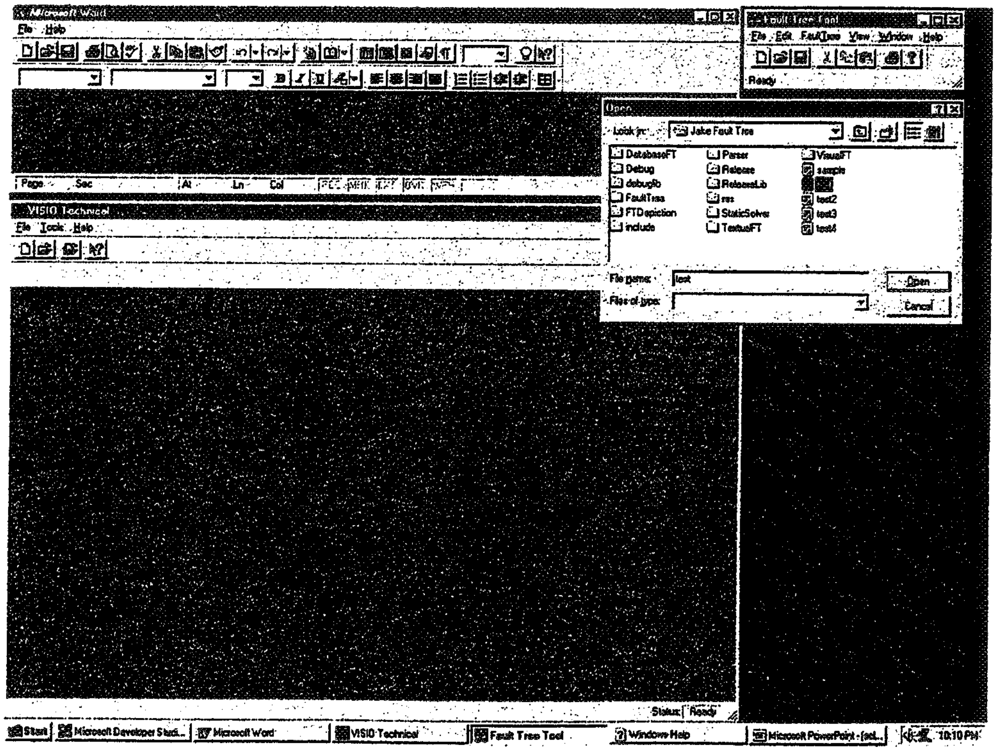
Workable Basis for Mega-Reuse

Evidence

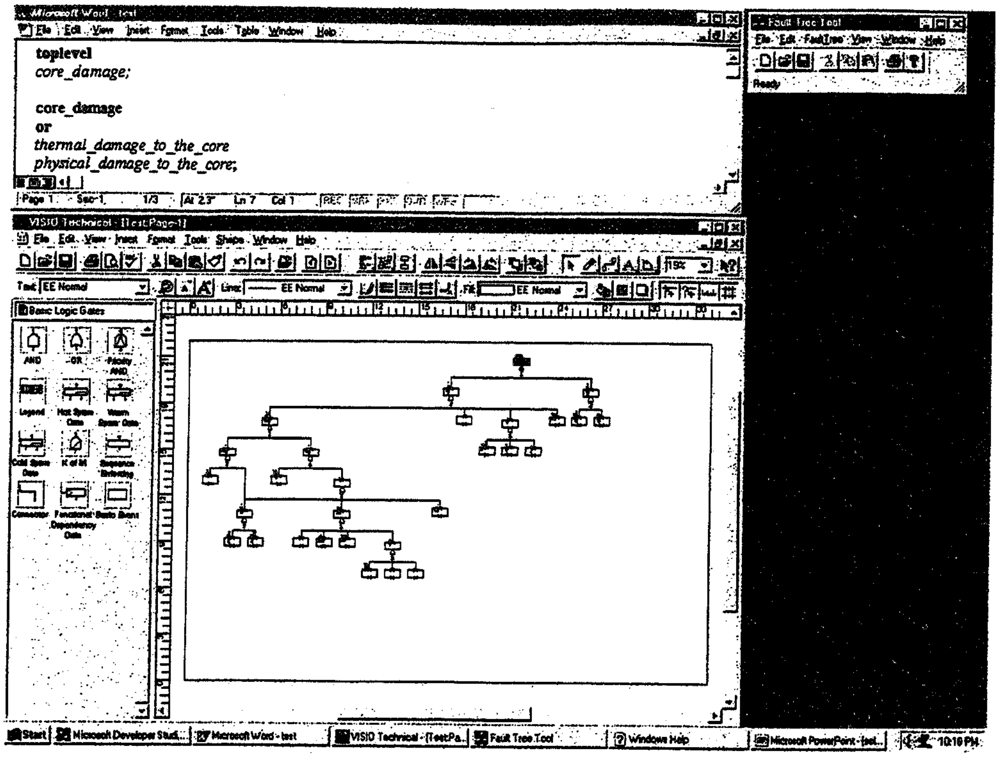
Appears to Work for Tools

Research Issue

Why?


Look in: Jake Fault Tree

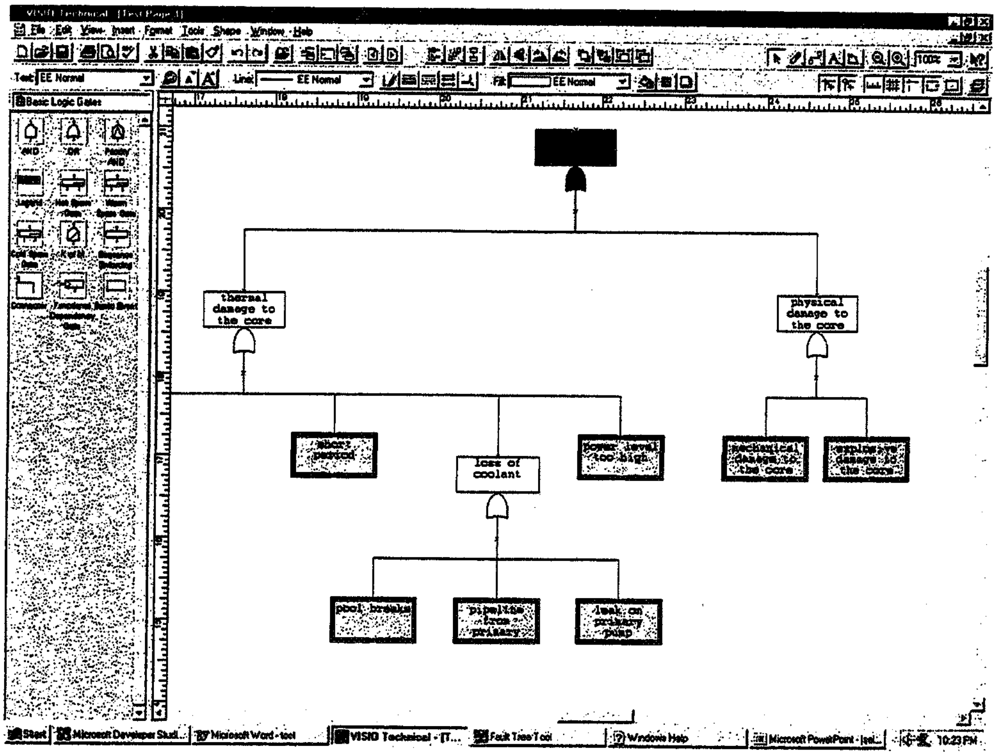
File name: test

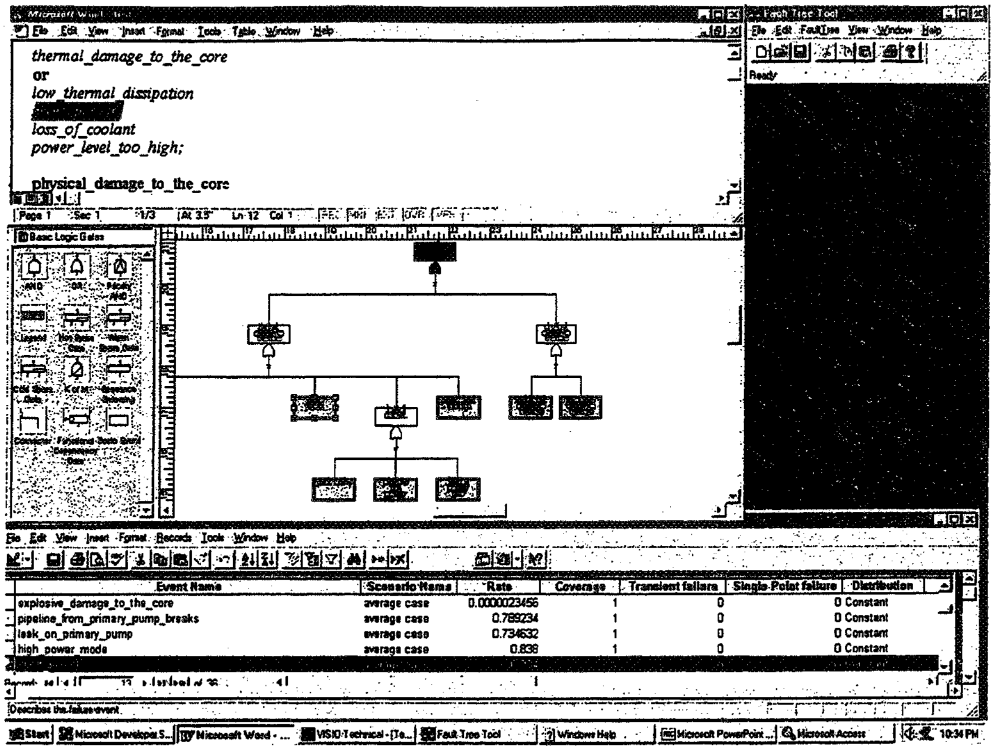


```
toplevel
core_damage;

core_damage
or
thermal_damage_to_the_core
physical_damage_to_the_core;
```

Microsoft Word - test
File Edit View Insert Format Tools Table Window Help
Page 1 Sec 1 1/3 At 2.3" Ln 7 Col 1
VISIO Technical - [TestPage-1]
File Edit View Insert Format Tools Shape Window Help
Basic Logic Gates
AND
OR
Priority AND
Legend
Hot Spare Gate
Warm Spare Gate
Cold Spare Gate
K of M
Sequence Enforcing
Connector
Functional Dependency Gate
Basic Event
Fault Tree Tool
File Edit FaultTree View Window Help
Ready
Start
Microsoft Developer Stud...
Microsoft Word - test
VISIO Technical - [TestPa...
Fault Tree Tool
Windows Help
Microsoft PowerPoint - [sel...
10:18 PM



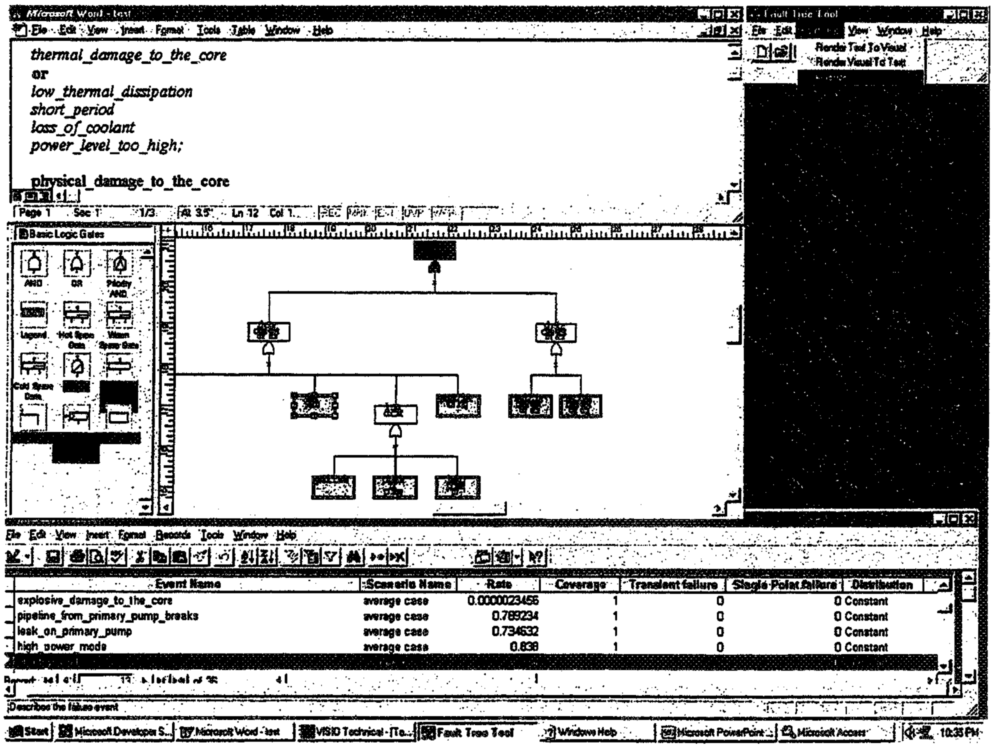


thermal_damage_to_the_core
or
low_thermal_dissipation
short_period
loss_of_coolant
power_level_too_high;

physical_damage_to_the_core

Event Name	Scenario Name	Rate	Coverage	Transient failure	Single Point failure	Distribution
explosive_damage_to_the_core	average case	0.0000023456	1	0	0	Constant
pipeline_from_primary_pump_breaks	average case	0.789234	1	0	0	Constant
leak_on_primary_pump	average case	0.734632	1	0	0	Constant
high_power_mode	average case	0.838	1	0	0	Constant

Describes the failure event.

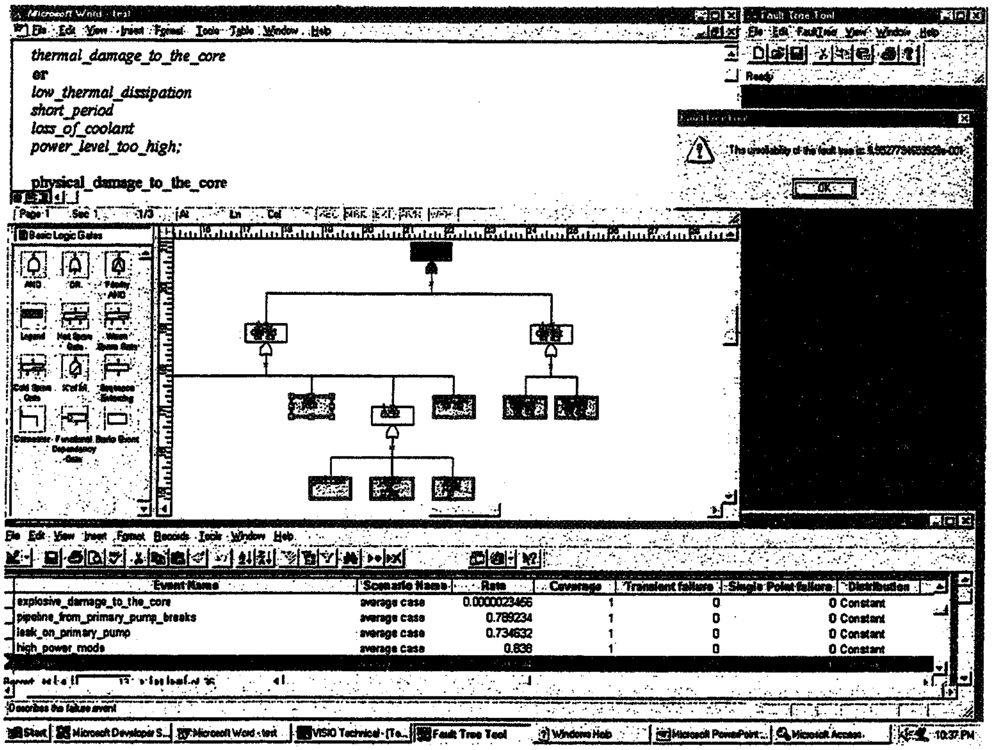


thermal_damage_to_the_core
or
low_thermal_dissipation
short_period
loss_of_coolant
power_level_too_high;

physical_damage_to_the_core

Event Name	Scenario Name	Rate	Coverage	Transient failure	Single Point failure	Distribution
explosive_damage_to_the_core	average case	0.0000023456	1	0	0	Constant
pipeline_from_primary_pump_breaks	average case	0.789234	1	0	0	Constant
leak_on_primary_pump	average case	0.734632	1	0	0	Constant
high_power_mode	average case	0.838	1	0	0	Constant

Describes the failure event



thermal_damage_to_the_core
or
low_thermal_dissipation
short_period
loss_of_coolant
power_level_too_high;

physical_damage_to_the_core

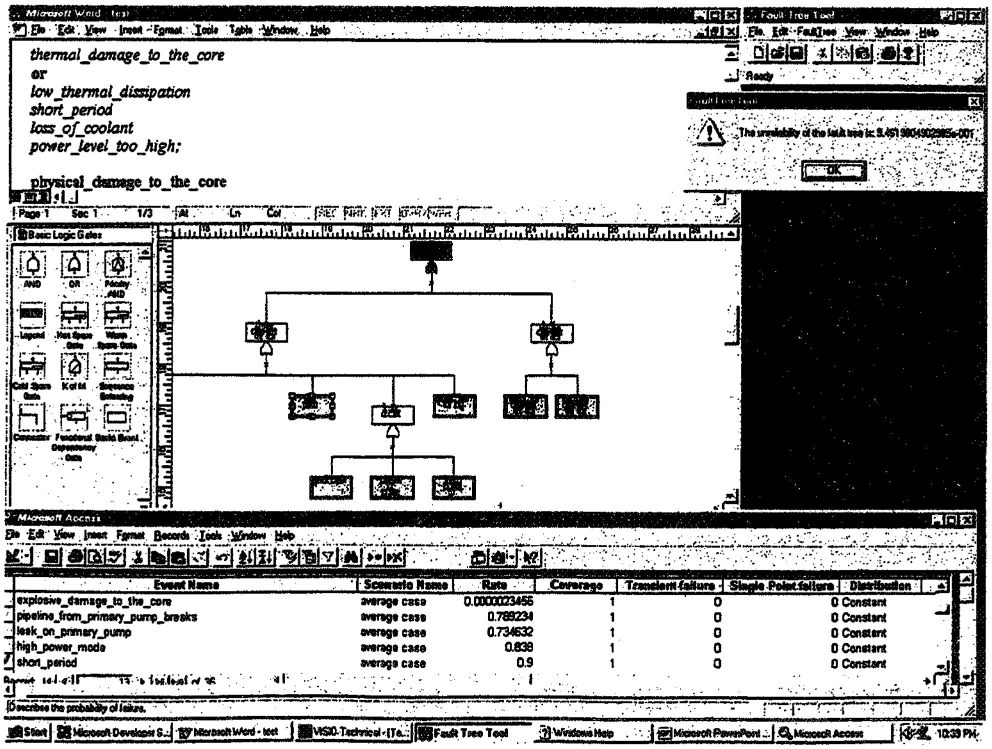
The unreliability of the fault tree is: 5.5527734603592e-001

OK

Event Name	Scenario Name	Rate	Coverage	Transient failure	Single Point failure	Distribution
explosive_damage_to_the_core	average case	0.0000023456	1	0	0	Constant
pipeline_from_primary_pump_breaks	average case	0.789234	1	0	0	Constant
leak_on_primary_pump	average case	0.734632	1	0	0	Constant
high_power_mode	average case	0.838	1	0	0	Constant

Record: 13 of 36

Describes the failure event



thermal_damage_to_the_core
or
low_thermal_dissipation
short_period
loss_of_coolant
power_level_too_high;

physical_damage_to_the_core

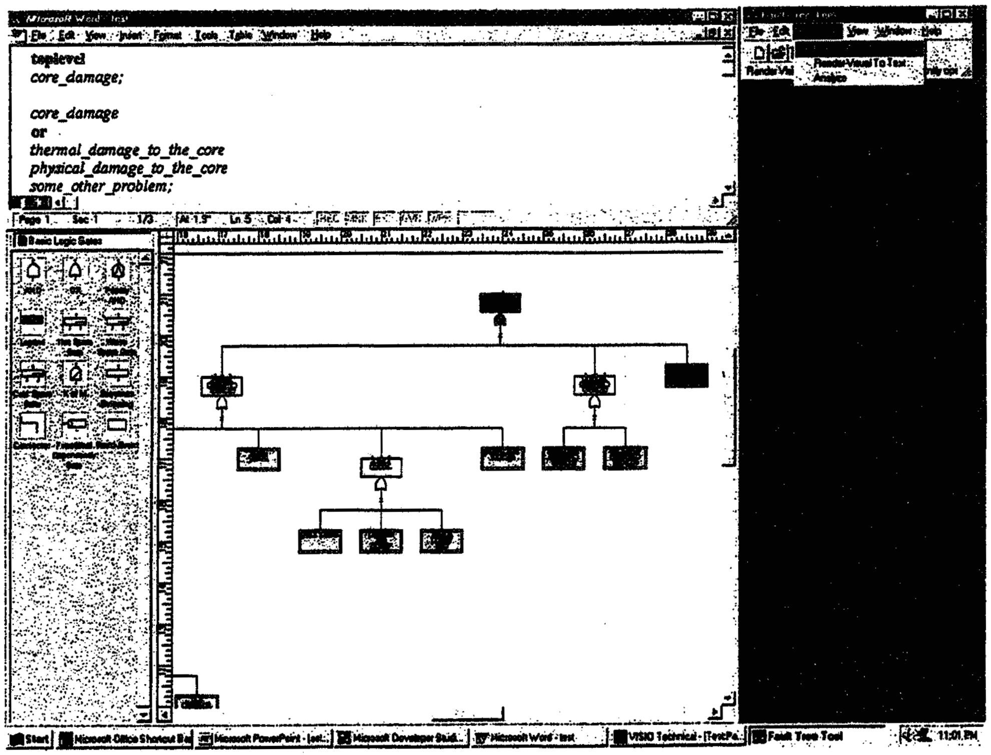
The unreliability of the fault tree is: 5.4519804902985e-001

OK

Event Name	Scenario Name	Rate	Coverage	Transient failure	Single Point failure	Distribution
explosive_damage_to_the_core	average case	0.0000023456	1	0	0	Constant
pipeline_from_primary_pump_breaks	average case	0.789234	1	0	0	Constant
leak_on_primary_pump	average case	0.734632	1	0	0	Constant
high_power_mode	average case	0.838	1	0	0	Constant
short_period	average case	0.9	1	0	0	Constant

Record: 13 of 36

Describes the probability of failure.



```
toplevel
core_damage;

core_damage
or
thermal_damage_to_the_core
physical_damage_to_the_core
some_other_problem;
```

Microsoft Word - test
File Edit View Insert Format Tools Table Window Help
Page 1 Sec 1 1/3 At 1.5" Ln 5 Col 4
Basic Logic Gates
AND
OR
File Edit View Window Help
Render Visual To Text
Analyze
Start
Microsoft Word - test
VISIO Technical - [TestPa...
Fault Tree Tool
11:01 PM

What Made It Work?

- ***Right Basis*** *Components*
- ***"MightyMorphic"*** *Components*
- ***High Valence*** *Components*

Right Basis For Tools

- Technical Drawing
- Text Management
- Data Management

- Domain-Specific Language
- Domain-Specific Types
- Domain-Specific Analysis

"MightyMorphism"

- Flexibility in Critical Dimensions
- High-level Specialization Mechanisms
- Provisions for Add-on Functionality
- Control Over User Interfaces

High Valence

- Architectural coherence (OLE)
- Application Programming Interfaces
- Managed object model (Visio)

Conclusion

- We Demonstrated Effective Mega-Reuse
- "Order of Magnitude Better Tool" --Dugan

- Promising Architectural Concept
- Investigation of Generalizability Warranted

TECHNOLOGY EVOLUTION: COTS Transition at Raytheon 1983-1996 (Part I)

59-61

415663

Tom Lydon, Laurie Fischer, Karl Gardner
Raytheon Company

360710

p28

ABSTRACT

The Raytheon RES Software Engineering Laboratory is a large software development organization consisting of about 1200 engineers. It has been independently rated as an SEI Level 3 site for four years and won the IEEE Software Process Achievement Award in 1995. The Raytheon process relies extensively on the use of integrated engineering tools to achieve process control. Data on 84 tools, 25 Raytheon-developed and 59 COTS, over the period 1983-1996 shows that the number of tools used in software engineering has grown from an average of about 4 tools per engineer in 1986 (not including standard host editors or compilers provided with the OS) to about 12 tools per engineer in 1996. Furthermore, over this period there was a definite, systematic swing from Raytheon-developed tools which were predominant from 1984-1990, to COTS tools which have been predominant since 1990. The current mix is about 3 Raytheon tools and 9 COTS tools in use per software engineer.

There are pros and cons to the use of COTS tools. Overall costs have initially gone up, but they are still expected to go down in the future, though this is not certain. Standardization is one method of controlling overall costs. Productivity and quality both appear to improve with the use of COTS tools, but this improvement data is also a result of other factors such as process initiatives, training, and better hardware (workstations). There are regular births and deaths of tools, and this "churning" must be managed. The data suggest that the overall use of tools will level off over the next few years at about 13 tools per engineer, 2 Raytheon and 11 COTS. COTS is not a panacea, but they are here to stay.

This paper is the initial portion of a study of overall COTS tool costs. Data on the use of Raytheon-developed and COTS tools is included, but life cycle cost data has not yet been collected. The second portion of this study will be completed early in 1997.

BACKGROUND

The Raytheon Electronic Systems (RES) Software Engineering Laboratory (SEL) is a large, diverse software development organization, geographically distributed across eight major sites in six different states (primarily Massachusetts). This laboratory develops software for the primary RES business areas of Command & Control Systems; Naval, Air to Air, and Strike Systems; Air Defense Battle Management and Radar Systems; THAAD/Ground Based Radar; and Transportation Systems, including Air Traffic Control.

The Raytheon RES Software Engineering Laboratory has been independently rated as an SEI Level 3 site for four years and won the IEEE Software Process Achievement Award in 1995. The Raytheon process relies extensively on the use of automated, integrated engineering tools to achieve process control. The number of software engineers in the current RES SEL has grown from about 600 in 1983 to about 1200 in 1996.

DATA ON TOOLS

During the 1980's, Raytheon had three major software laboratories located at Missile Systems Division in Bedford and Burlington MA, at Equipment Division in Sudbury, Wayland, and Marlboro MA, and at Submarine Signal Division in Portsmouth RI. These divisions employed about 600 software engineers in early 1983. The three divisions were consolidated into a single division, Raytheon Electronic Systems (RES) in early 1995, and three software laboratories are now consolidated into a single Software Engineering Laboratory (SEL).

In the 1980's, there were three types of tools used:

(a) Raytheon-developed internal tools
(b) Tools provided "free" with operating systems, such as yacc, lex, lint, curses, and troff with Unix, and CMS with VMS
(c) Purchased third-party tools, now known as COTS

Raytheon's policy was always to encourage purchasing category (c) wherever possible, in preference to building our own in category (a), but the fact was that most of our requirements could not be met by COTS tools, so the majority of tools were developed internally. The data included in this study is from categories (a) and (c), and does NOT include data on the use of tools that were provided standard as "no cost" components of the operating system.

Also in the 1980's, there were four sources of funding for tool acquisition or development:

SOURCE	Approx % of Funding	- COMMENT
Corporate	40%	- interdivisional initiative to aide many projects - used mainly for tool development
Cost Center	30%	- common tools used by many programs - costs centrally absorbed, redistributed to programs
Program	20%	- used for tools specific to program needs - tools owned by the program, not Raytheon
Overhead	10%	- used for productivity improvement tools - used for special tools on high-end computers

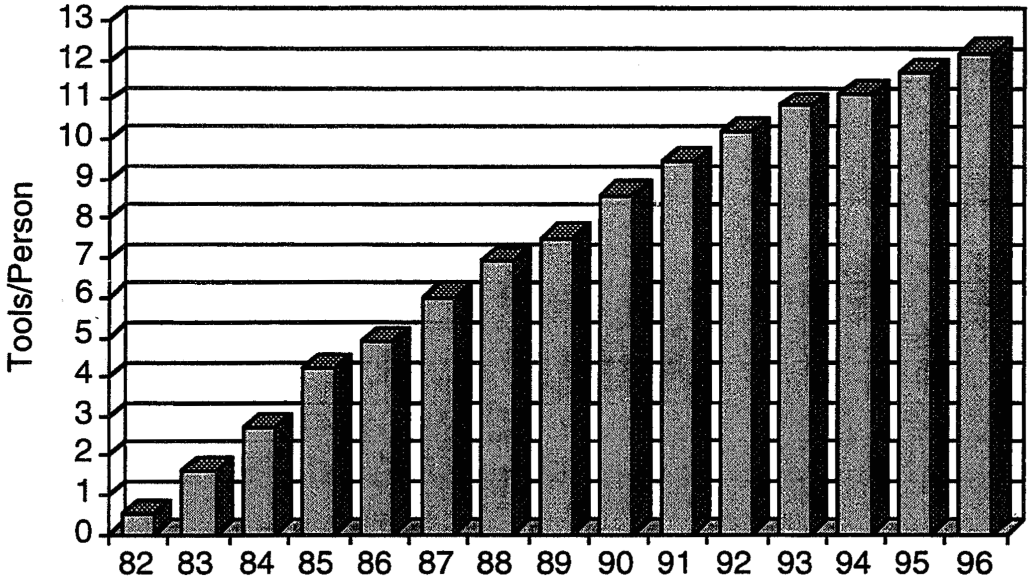
As lead engineer for both Corporate and Cost Center tool programs at the time, and through regular interaction with programs and overhead tasks, I was able to reconstruct good historical data on the extent of use of 84 tools over the period 1983-1996. The goal was to analyze the true costs of conversion from internal to COTS tools. Data on tool use has been developed, but data on costs has not yet been finalized.

This study of 84 tools includes 25 Raytheon-developed tools and 59 COTS, mostly for computer-aided software engineering (CASE). The data is shown below in a table where each row represents one tool, and the columns represent average "Fraction-of-Use" data for each year. "Fraction-of-Use" is the decimal fraction of software laboratory engineers who used the tool on a regular basis during the year. For example, ".3" means that 30% of the engineers used that particular tool (either internal or COTS) on a regular basis that year, so if there were 800 total engineers then 240 used the tool.

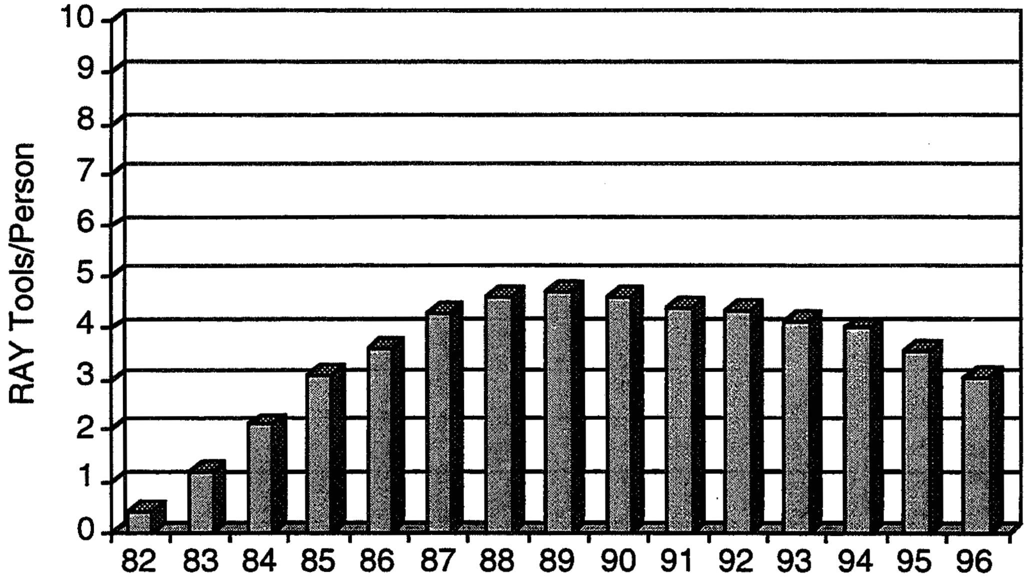
TYPE	CATEGORY	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	
COTS	REQTS	0.1	0.1														Niche
RAY	REQTS	0.1	0.2	0.2	0.1												Niche
RAY	CM	0.1	0.2	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.2	0.2	0.2	0.1	0.1	0.1	Common
RAY	DOCUM	0.2	0.3	0.5	0.3	0.1											Common
COTS	COST		0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.05	0.01	0.01	0.01	0.01	0.01	Niche
COTS	COST		0.1	0.1	0.1	0.1											Niche
COTS	CODING		0.1	0.1	0.1	0.1											Niche
RAY	DESIGN		0.1	0.2	0.4	0.4	0.4	0.4	0.4	0.3	0.2	0.1	0.1	0.05	0.05	0.01	Common
RAY	DESIGN		0.1	0.2	0.3	0.4	0.4	0.4	0.3	0.3	0.2	0.1	0.1	0.05	0.05	0.01	Common
RAY	CM		0.1	0.2	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.5	0.4	0.4	0.2	0.1	Common
RAY	MAINT		0.2	0.3	0.3	0.2	0.2	0.1	0.1								Common
RAY	MAINT			0.1	0.1	0.1											Niche
COTS	DB			0.1	0.2	0.1											Niche
RAY	MGMT			0.1	0.3	0.5	0.5	0.4	0.3	0.2	0.1						Common
COTS	DATA			0.2	0.4	0.5	0.5	0.4	0.2	0.1	0.1						Common
RAY	DESIGN				0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.2	0.2	0.1	0.01	Common
RAY	CM				0.1	0.2	0.3	0.4	0.4	0.5	0.5	0.5	0.4	0.4	0.4	0.3	Common
COTS	DB				0.1	0.2	0.3	0.4	0.5	0.5	0.6	0.6	0.6	0.5	0.4	0.3	Common
COTS	DB				0.1	0.2	0.3	0.4	0.5	0.5	0.6	0.6	0.6	0.5	0.5	0.4	Common
RAY	DEFECT				0.1	0.1	0.2	0.3	0.3	0.4	0.4	0.4	0.4	0.4	0.4	0.4	Common
RAY	MAINT				0.1	0.2	0.4	0.4	0.3	0.2	0.1						Common
RAY	DEFECT				0.1	0.1	0.2	0.3	0.4	0.4	0.5	0.5	0.5	0.5	0.5	0.4	Common
RAY	DOCUM				0.2	0.4	0.6	0.6	0.5	0.4	0.3	0.2	0.1				Common
RAY	DATA					0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.3	0.3	Common
COTS	DATA						0.1	0.1	0.1	0.2	0.2	0.2	0.2	0.2	0.2	0.3	Niche
COTS	DOCUM						0.1	0.2	0.1								Niche
COTS	DOCUM						0.1	0.1	0.2	0.2	0.1	0.1	0.1	0.05	0.05	0.01	Niche
COTS	MGMT						0.1	0.1	0.1	0.1	0.1	0.05	0.05	0.01			Niche
COTS	CODING						0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.3	0.3	Common
RAY	TEST							0.1	0.1	0.1	0.2	0.2	0.2	0.2	0.2	0.2	Niche
COTS	DOCUM							0.1	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	Common
COTS	DESIGN							0.1	0.1	0.2	0.2	0.1	0.2	0.2	0.3	0.3	Common
COTS	REQTS							0.1	0.2	0.3	0.2	0.1	0.1				Common
COTS	DATA							0.1	0.2	0.4	0.6	0.7	0.7	0.8	0.8	0.8	Common
RAY	TRACE								0.1	0.1	0.1	0.2	0.2	0.2	0.1	.0.1	Niche
RAY	DEFECT								0.1	0.2	0.3	0.3	0.3	0.4	0.3	0.3	Common
COTS	DOCUM								0.1	0.3	0.6	0.7	0.8	0.8	0.8	0.7	Common
COTS	DESIGN								0.1	0.3	0.3	0.3	0.3	0.3	0.3	0.3	Common
RAY	TRACE									0.1	0.1	0.2	0.2	0.3	0.3	0.2	Common
COTS	DATA									0.1	0.2	0.3	0.2	0.2	0.2	0.2	Common
COTS	GUI									0.1	0.1	0.1	0.1	0.1	0.1	0.2	Niche
COTS	MGMT									0.1	0.2	0.3	0.3	0.4	0.5	0.6	Common
COTS	CODING									0.1	0.1	0.1	0.1	0.2	0.2	0.2	Niche
COTS	DATA										0.1	0.1	0.1	0.1	0.1	0.1	Niche
COTS	DATA										0.1	0.1	0.1	0.1	0.1	0.1	Niche
RAY	DEFECT										0.1	0.2	0.3	0.3	0.3	0.2	Common
COTS	GIS											0.05	0.1	0.05	0.01	0.01	Niche
COTS	GUI											0.1	0.1	0.1	0.1	0.1	Niche
COTS	DATA											0.1	0.1	0.1	0.1	0.1	Niche
COTS	DESIGN											0.1	0.1	0.2	0.2	0.3	Common
RAY	MGMT											0.1	0.1				Niche
COTS	CM											0.1	0.2	0.3	0.5	0.8	Common
COTS	CM											0.1	0.1				Niche
COTS	DOCUM											0.1	0.1	0.1	0.05	0.01	Niche
RAY	DATA											0.05	0.05				Niche
COTS	DB											0.05	0.05	0.05	0.01	0.01	Niche
COTS	COST												0.05	0.05			Niche
COTS	COST												0.05	0.05	0.05	0.05	Niche
COTS	GIS												0.01	0.01			Niche
COTS	GUI												0.1	0.1	0.1	0.1	Niche
COTS	DATA												0.1	0.1	0.1	0.1	Niche
COTS	CODING												0.1	0.2	0.3	0.4	Common
COTS	DOCUM												0.1	0.2	0.1		Niche
RAY	DEFECT												0.1	0.2	0.2	0.1	Niche
COTS	REQTS													0.01	0.01		Niche
COTS	REQTS													0.1	0.1	0.1	Niche
COTS	DB													0.05	0.1	0.1	Niche
COTS	DOCUM													0.05	0.1	0.1	Niche
COTS	DOCUM														0.1	0.1	?
COTS	GUI														0.05	0.05	?
COTS	CODING														0.1	0.1	?
COTS	CODING														0.1	0.1	?
COTS	CODING														0.1	0.1	?
COTS	CODING														0.1	0.2	?
RAY	DEFECT														0.1	0.3	Common
COTS	CM														0.1	0.1	?
COTS	DOCUM															0.01	?
COTS	TRACE															0.01	?
COTS	DEFECT															0.01	?
COTS	DESIGN															0.1	?
COTS	TEST															0.1	?
COTS	DOCUM															0.1	?
COTS	DATA															0.1	?
COTS	CODING															0.1	?
TOTAL	84	0.5	1.6	2.7	4.2	4.9	6	6.9	7.5	8.6	9.45	10.2	10.9	11.2	11.7	12.2	
RAY	25	0.4	1.2	2.1	3.1	3.6	4.3	4.6	4.7	4.6	4.4	4.35	4.15	4	3.6	3.03	
COTS	59	0.1	0.4	0.6	1.1	1.3	1.7	2.3	2.8	4	5.1	5.9	6.7	7.2	8.1	9.2	

The data is usually rounded to the nearest 10% due to inability to measure the data more precisely. For example, some projects used tools for parts of the year, and other projects did not have 100% of their engineers use a given tool at a given time, so annual weighted average Fractions-of-Use have been totaled across all programs and rounded to the nearest tenth. In some cases, small numbers such as .05 (5%) or .01 (1%) are used to indicate that a tool was still in active use, but only by small populations of users.

- **Overall Use of Software Tools has Steadily Increased**

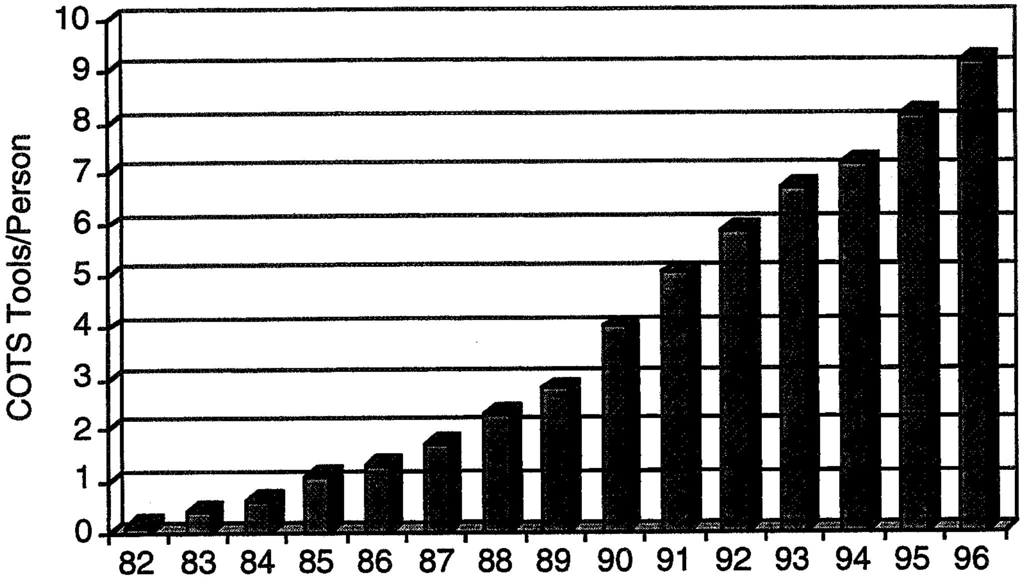


- **Use of Raytheon-Developed Tools has Dropped Off**

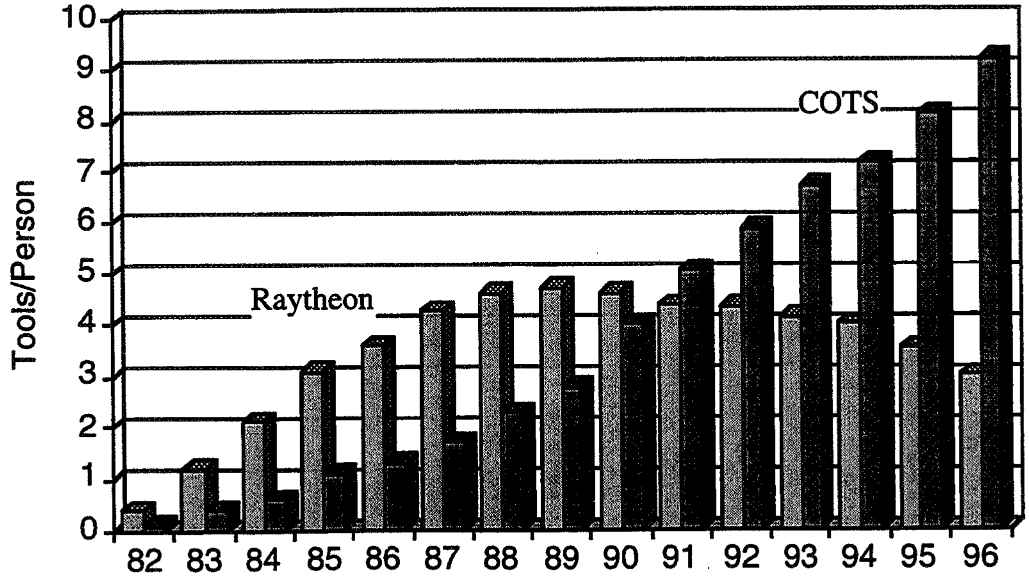


The data shows a definite, systematic swing from Raytheon-developed tools to COTS tools. As the use of COTS CASE tools has increased, the use of comparable Raytheon-developed tools has declined, while the overall use of tools for software engineering has steadily increased.

- **Use of COTS Tools has Increased Dramatically**



- **Counterpoint: Swing from Internal to COTS Tools**



The shift from internally developed to COTS software tools over this period has been driven by several factors, including:

1. Customer Requirements - Customers are becoming increasingly knowledgeable and sophisticated in their requirements for software development. In some cases, they require certain specific tools to be used on a program. In other cases, they require a COTS tool be used, if not a specific tool.

2. Need to Improve Productivity - Contractors are in constant competition, and if there is a better/faster way to develop software, they must learn and take advantage of it. Many COTS tools embody well-documented computer-aided software engineering (CASE) methodologies specifically aimed at improving productivity.

3. Need to Improve Quality - Similar to productivity, competition for improved software quality (fewer defects) is acute. COTS tools with CASE methodologies also specifically aim at improving quality.

4. Need to Improve Turnaround Time - Sometimes called time-to-market, this is often more important the either productivity or quality. A task may cost the same, or it may cost more, but if you can complete it with the same quality in half the time there is often a premium that can be gained.

5. Need to Reduce Costs - In-house tools require internal staffing for maintenance, upgrades, and support. COTS tools appear now to be mature enough that vendors can get a wide enough usage base to defer these costs more cost-effectively than any one user could do themselves. Thus there is an opportunity to reduce internal staff (or redirect to other projects) and reduce overall computing costs to programs.

6. Standardization/Integration - By using COTS tools, it is easier to standardize tools across organizations (e.g. divisions), and to provide more standard tool integration mechanisms, allowing for synergy across tools and programs.

The primary development environment(s) for Raytheon software development have evolved gradually over the past 15+ years. In the 1980s there were almost equal amounts of VMS and Unix based development. From 1990-1995 it was primarily Unix based (several flavors) development. Since 1995 it has still been primarily Unix development, but we now see many smaller, more commercial programs beginning to use NT as the platform of choice.

TYPES OF TOOLS

The data on the 84 tools can be divided into the following categories of tools, by primary application to the software life cycle:

CM	Configuration Management, code control, automated build tools (internal tools have been built on top of sccs, RCS, and CMS)
CODING	Support for coding, compilers, debuggers, environments
COST	Cost estimation tools
DATA	Data analysis and data reduction tools
DB	Database related tools, e.g., relational or object-oriented tools
DEFECT	Defect collection, tracking, reporting tools
DESIGN	Preliminary and detailed design tools, PDL and graphical
DOCUM	Documentation tools
GIS	Geographic Information Systems tools
GUI	Graphical User Interface tools
MAINT	Maintenance and code documentation tools
MGMT	Management tools
REQTS	Requirements analysis tools (some overlap with prelim. design)
TEST	Test support, test generation, test tracking tools
TRACE	Traceability tools

The mix of tools by category and type (Raytheon or COTS) is shown below:

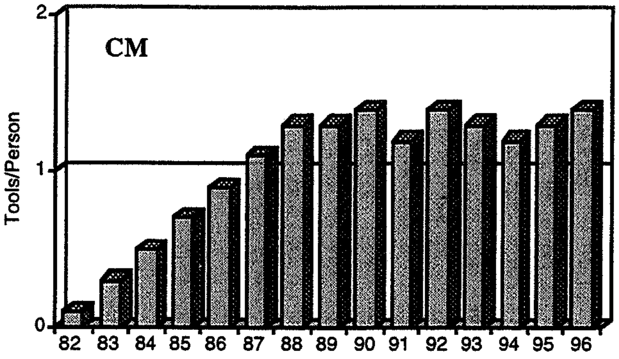
Category	**Raytheon**	**COTS**	**TOTAL**
CM	3	3	6
CODING	0	9	9
COST	0	4	4
DATA	2	9	11
DB	0	5	5
DEFECT	6	1	7
DESIGN	3	4	7
DOCUM	2	10	12
GIS	0	2	2
GUI	0	4	4
MAINT	3	0	3
MGMT	2	2	4
REQTS	1	4	5
TEST	1	1	2
TRACE	2	1	3
	25	**59**	**84**

Raytheon is a large software engineering organization that is NOT primarily a developer of software tools. From the distribution of data, it is clear the Raytheon has spent relatively more effort on defect-tracking and maintenance tools, and relatively little effort on coding, cost estimation, data analysis, database, documentation, GIS, and GUI tools.

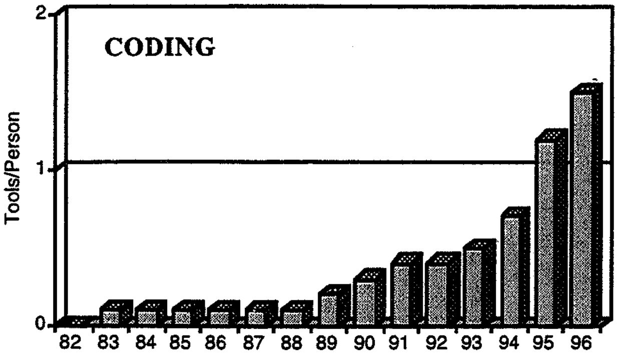
INDIVIDUAL TOOL CATEGORIES

A closer look at a few tool categories reveals some micropatterns in the data, for example:

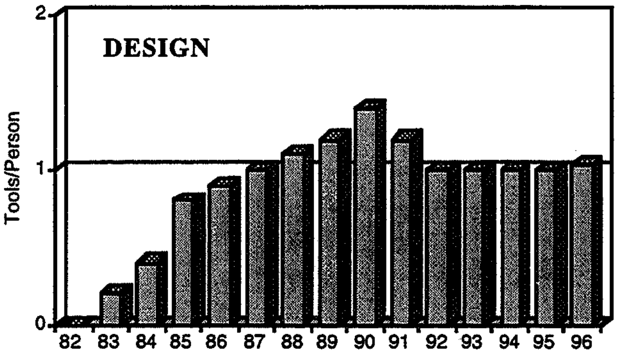
a) The use of CM tools increased linearly until about 1987 when essentially all engineers used a CM tool on their project. Since then it has leveled off at about one tool per engineer, as would be expected. (The amount over one may be due to data rounding up.)



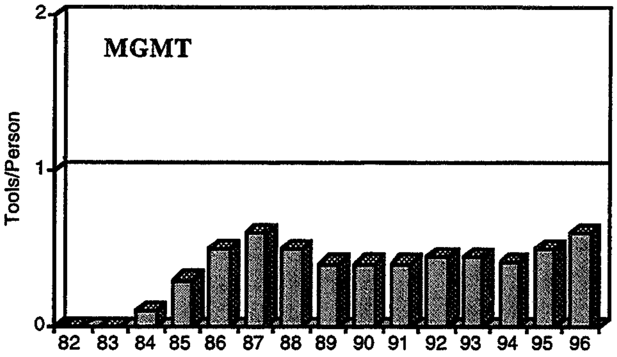
b) The use of CODING tools increased very slowly through the 1980s, probably due to an acceptable level of "standard" operating system-supplied tools. The number of CODING tools has increased dramatically since 1993, however, indicating an active need for integrated coding environment tools, especially for embedded systems development.



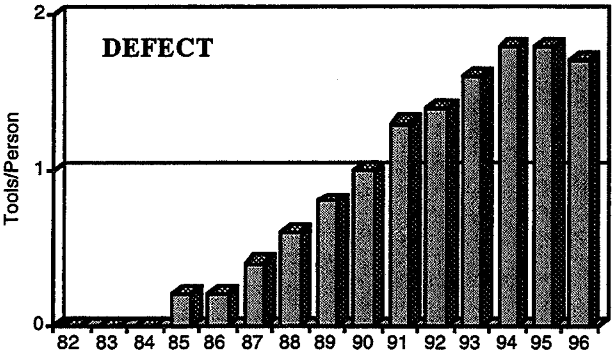
c) The use of DESIGN tools increased quickly to about one per engineer in the mid-1980s, probably due to immediate productivity and quality gains from the use of these tools. As expected, it has leveled off at one tool per engineer, since it is rare that a project would need to actively use two different, independent DESIGN tools per engineer.



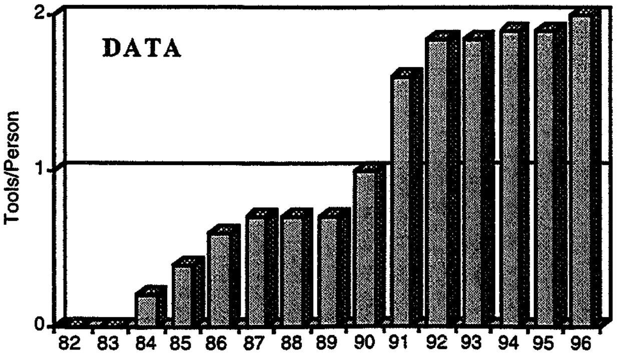
d) The use of MANAGEMENT tools has remained at about one-half tool per engineer since the mid-1980s. This makes sense, since not all engineers need to prepare schedules, track actuals, and report on project status. Group leaders, software lead engineers, section managers need to use these tools and make up about 1/3 the engineering base.



e) The use of DEFECT tools has risen steadily since the mid-1098s. This is partly because there has been a variety of tools, with projects having their own preferences, sometimes their own home-grown tools, and no consistent standard across all projects.



f) The use of DATA tools blossomed initially in the mid-1980s, and again in the early 1990s. The fist wave was probably due to the need for basic data analysis capability, but the second wave is more likely due to increased an emphasis on quantitative data collection and management while going from SEI CMM Level 3 to Level 4, which is what is happening at Raytheon now.



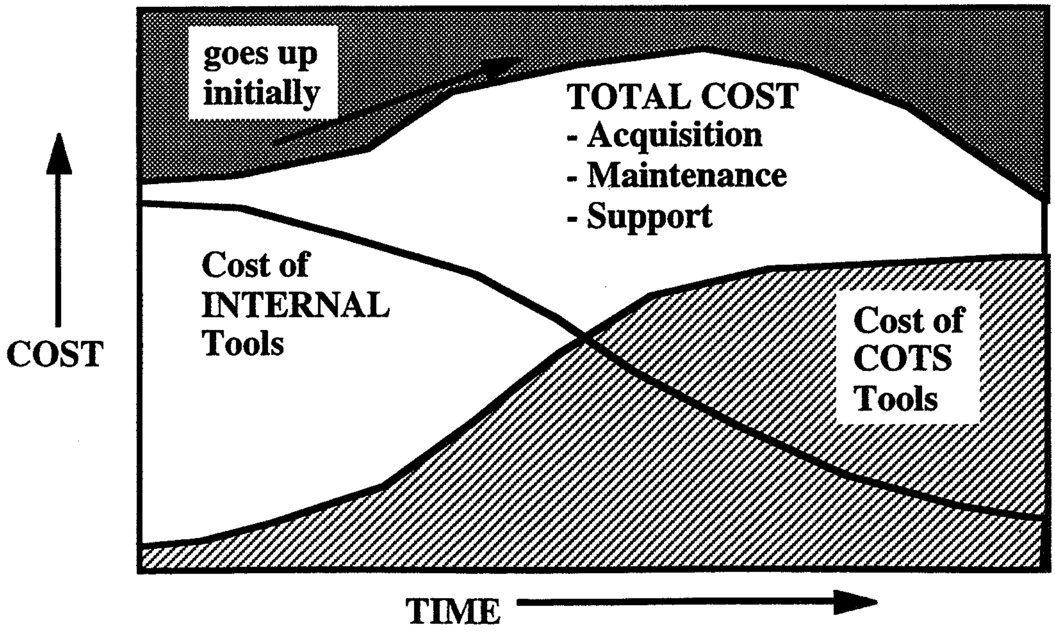
COTS TOOL ARE NOT A PANACEA

Despite the well-documented shift to using COTS tools in place of internal tools, there is a new set of problems that must be dealt with due to this transition.

The main tradeoff for a large software contractor is gaining quality tools at a reduction in cost (at least that's the promise) versus giving up control over those tools. The table below illustrates the principal PROS and CONS of this tradeoff:

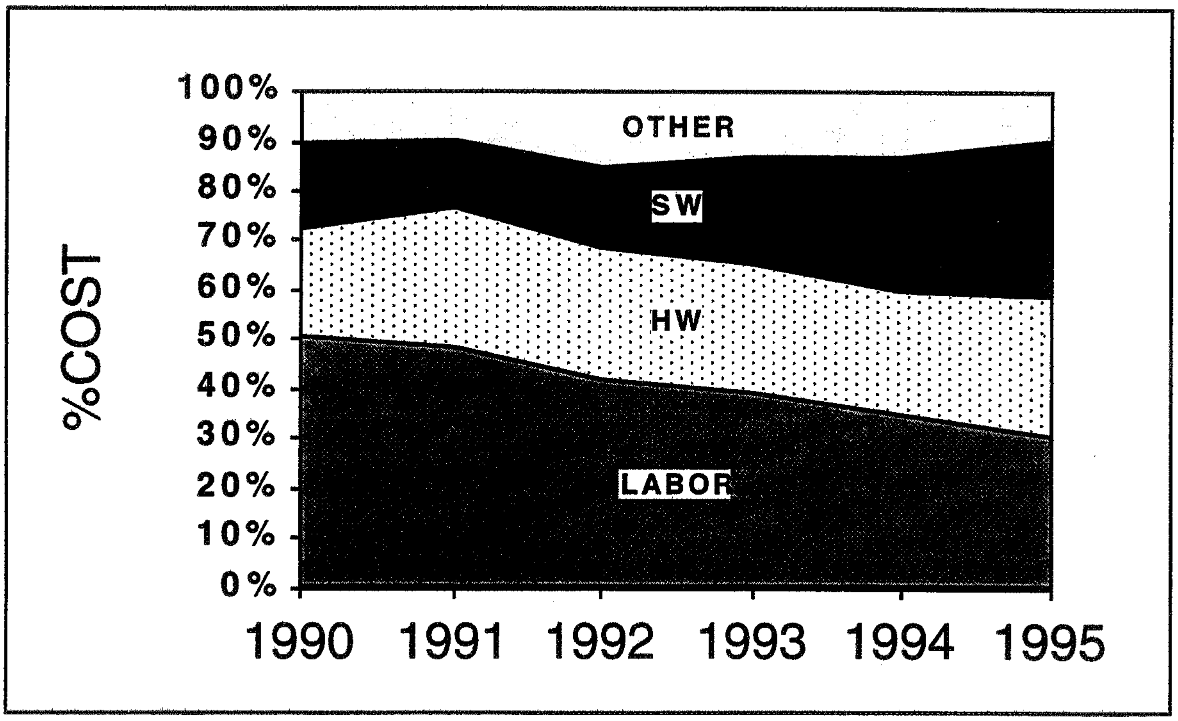
	PROS	CONS
Internal Tools	• Cost is fixed, not proportional to #users • Problem fixes usually faster • New capabilities can be added faster	• More expensive to develop • More manpower required for maint & support • Usually lower quality tools • Slower initial availability
COTS Tools	• Less expensive to develop • Less manpower required for maint & support • Usually better quality tools • Faster initial availability	• Cost increases almost linearly with #users • Problem fixes usually slower • New capabilities usually not added quickly

Instead of immediately lowering costs, during transition from internal to COTS tools the costs appear to initially go up, due to the startup costs of tool acquisition and the fact that legacy tools must still be supported for some period of time on ongoing projects that cannot afford to switch mid-stream. A schematic diagram of this short term cost "bubble" effect is shown below:



There are other indicators of a more general, more persistent shift of support costs from hardware and labor to software, however. In a review of the costs incurred by the cost center to support about 500 software engineers in the former Missile Systems Division between 1990-1995, the percentage of costs related to software (purchase, maintenance, amortization) doubled from about 15% to about 30%, while labor fell from 50% to 30%, and hardware (purchase, maintenance, depreciation) remained constant at about 25-30% of total support costs. Other factors (supplies, allocations, etc) accounted for the remaining 10%.

Software maintenance was the highest growth factor, since it is a function not of purchases, but of total active inventory. These trends are shown in the figure below:

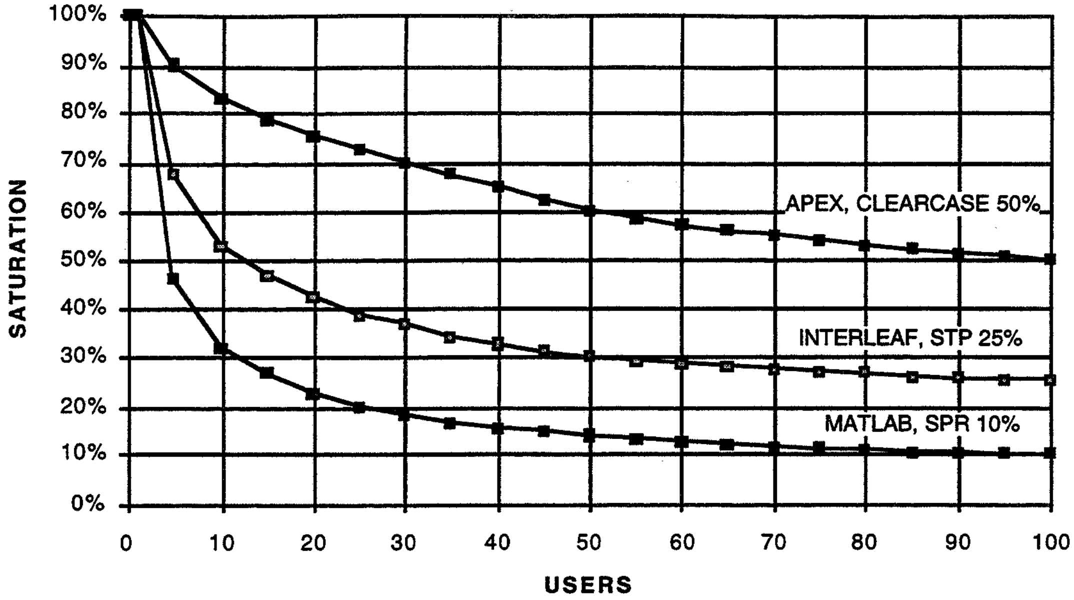


One way to help control the number of COTS licenses required, and thus the cost, is to consolidate concurrent licenses as far as possible onto large license servers. This is not always possible, for example on classified projects, but there is an economy of scale on licenses required related to an increasing total number of users.

For example for tools such as Rational Apex or Atria Clearcase, 10 networks of 10 users each would require 8-9 licenses on each network for a total of about 85 concurrent licenses, whereas 100 users on a single network would require only about 50 licenses, saving 35 licenses or 40% up front.

We have observed three classes of tools with different economies of scale based on usage patterns, referred to here as High, Medium, and Low Saturation tools. "Saturation" is the number of licenses required as a percent of the total number of users, and reflects how intensively a tool is used by an engineer. Coding environments tend to be high saturation tools, since coders usually stay in them most of the day, while occasional use tools tend to be low saturation tools.

- **For large networks, fewer licenses are required per user**



- HIGH Saturation tools settle at about 50% licenses per users (e.g. Apex, Clearcase)
- MEDIUM Saturation tools settle at about 25% licenses per users (e.g. Interleaf, StP)
- LOW Saturation tools settle at about 10% licenses per users (e.g. MatLab, SPR)
  (Note: SPR is a Raytheon-developed internal problem reporting tool)

STANDARDIZATION

Another way to minimize the cost of tools, especially COTS, is to maintain a standard list of supported software. Raytheon currently maintains a list of 22 standard tools that are supported by the main software engineering cost center. This provides a financial incentive for programs to use the same tools for design, code, test, CM, documentation, etc., without "forcing" them to do so. This is important because there are cases where a program is REQUIRED to use a particular tool specified by the contract or by the customer (or PREFERS a non-standard tool), and they are able to do this by paying for the software out of program funds without impacting other programs who do not need the tool. Conversely, if a program can opt to use a "standard" tool, it will incur no extra cost over the normal charge to use other cost center software.

This Standard Software List has encouraged the use of a minimum common set of tools (sometimes more than one tool per category, however), economies of scale in centrally serving licenses, in consolidating training, and in purchasing ability (minimum number of suppliers, maximum leverage), without unnecessarily restricting programs.

The number of tools on the standard list has varied since the list was started in 1994. It increased from 18 in 1994 to 26 in 1995, due to the consolidation of the three divisions and several cost centers into one, and the need to expand the standard list based on current tool use across all sites. These 26 were reduced to 22 in 1996, in an effort to streamline and further control overall costs.

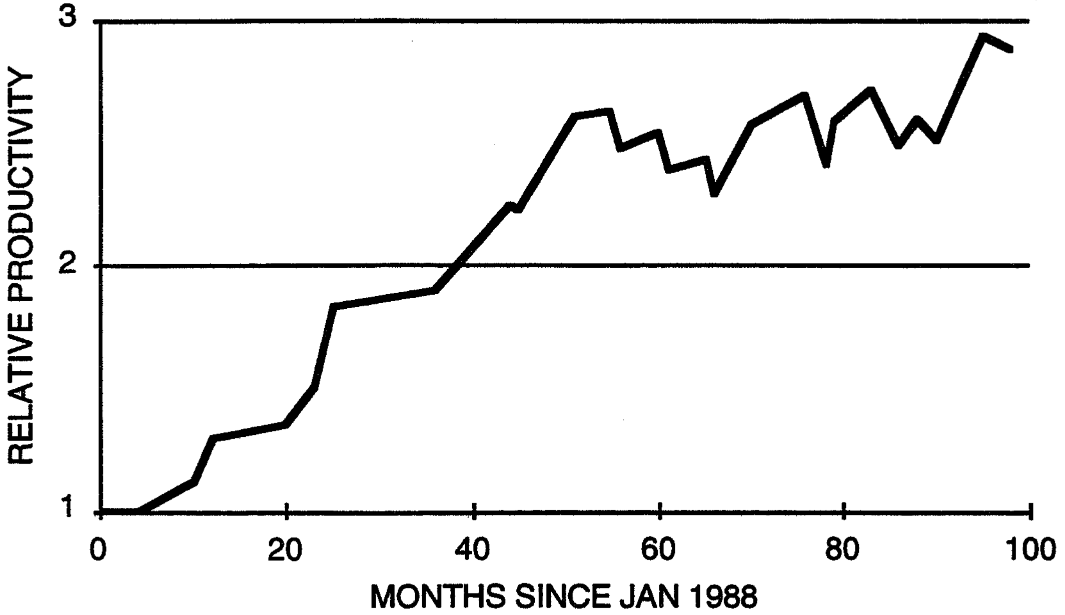
IMPROVED PRODUCTIVITY AND QUALITY

Raytheon has measured changes in organizational productivity and quality over the past 8+ years, and the trends show continuously productivity and quality improving simultaneously. The exact impact of increased use of software tools (or COTS tools) is hard to extract from the data, however, since there were several factors at work at the same time. These factors included:

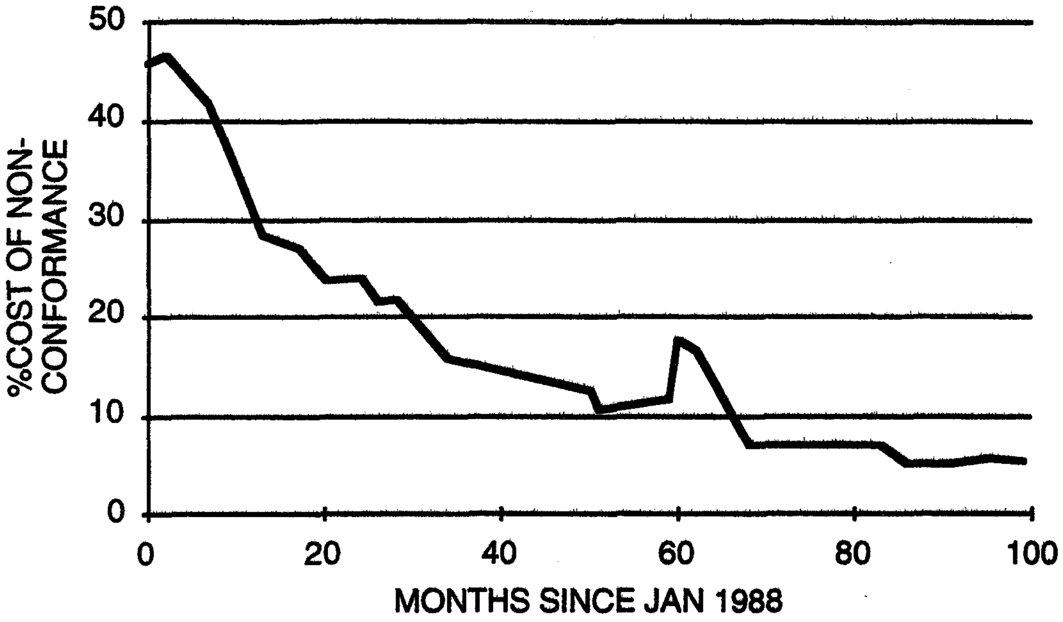
- hardware improvement (workstations, servers, networking, etc)
- process improvement (standards, policies, procedures, inspections, etc)
- and expanded training
- in addition to the increased use of tools

This was the same time period that the Raytheon software laboratories went from SEI Level 1 to Level 3, achieved ISO 9001, documented overall process improvement savings (Ref. Ray Dion), and were awarded the IEEE Process Award. It is clear that better quality software was an important contributor, but not the only contributor, to this improvement. The two figures below show DSI/MM productivity improvement (Jan 1988 is normalized to 1.00), and quality improvement as measured by reduced Cost of Nonconformance (cost of rework), which can be considered a proxy for the number of defects.

- **Tools help improve productivity**



- **Tools help reduce defects & rework**



OTHER OBSERVATIONS

A closer empirical look at the data reveals some other interesting observations:

- The total number of UNIQUE TOOLS in use at one time has increased from 4 to 62, however, (a) the measure of 4 in 1982 is incomplete and probably closer to 6 or 8, and (b) 14 of the 62 currently active are "almost dead" (very few users, usually legacy), leaving 48 truly currently active

- The number of DEATHS (discontinued use of a tool) over this period was 22, or an average of about 2 per year. Another observation is that tools are often slow to "die", as projects gradually discontinue use, but a few keep using them until the projects end.

- The average LIFESPAN of the 22 tools that "died" was 4.4 years.

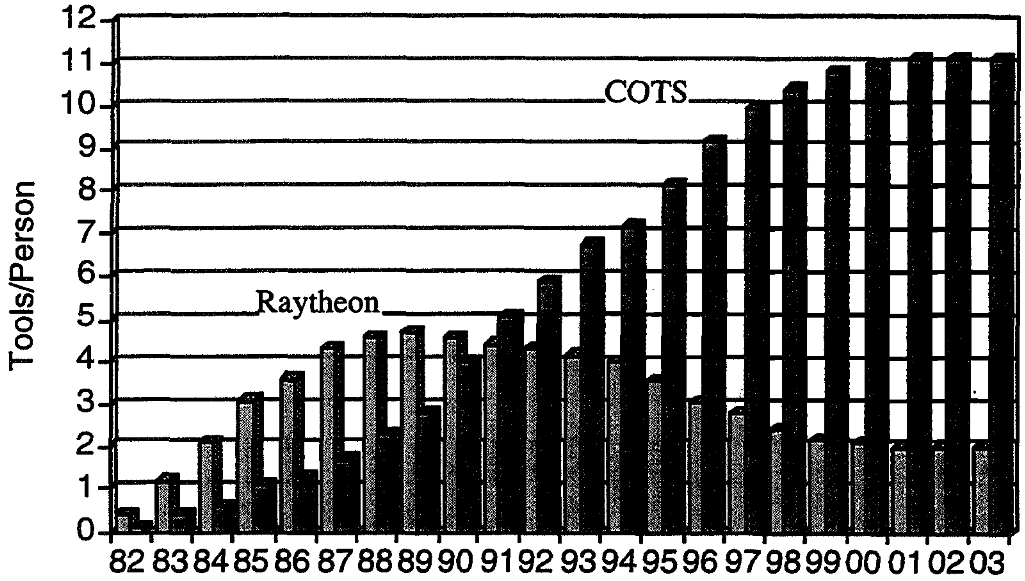
- The percentage of tools that reached COMMON status (that is, reaching more than 25% of the engineers in the software lab) is 39%

- The percentage of tools that never exceeded NICHE status (that is, never reaching 25% of the engineers in the software lab) was 43%

- For the remaining 18% of the tools it is too soon to tell whether they will become COMMON tools, or will remain NICHE tools

FUTURE STABILIZATION

From the data we've analyzed, and from observation of tool use by software engineers on many programs, it appears that the increase in the number of software tools in use may level off in the next few years at around 13 per engineer (about 2 Raytheon and about 11 COTS), though it is too soon to determine this with confidence. This projection is shown in the figure below:



OTHER LESSONS LEARNED

- THE TRANSITION TO COTS TOOLS IS PERMANENT
  - ALTHOUGH COTS TOOLS ARE NOT A PANACEA

- THERE IS A SHORT-TERM INCREASE IN COST, THOUGH (HOPEFULLY) A LONG-TERM COST DECREASE --> THE QUESTION IS: HOW MUCH??

- TO CONTROL COSTS, SUPPORT "STANDARD" TOOLS WITH $ INCENTIVE
  - THESE CAN BE SUPPLIED BY A COST CENTER FOR "FREE"

- TOOLS CAN HELP IMPROVE PRODUCTIVITY & QUALITY
  - EXACT AMOUNT IS HARD TO DETERMINE; THERE ARE OTHER FACTORS

- EXPECT A REGULAR "CHURNING" OF THE EXACT TOOL MIX
  - THERE WILL BE REGULAR BIRTHS AND DEATHS (ABOUT 2-3 PER YEAR)
  - THERE ALWAYS SEEMS TO BE (AT LEAST) TWO OF EVERY TYPE OF TOOL

- THE OVERALL BALANCE OF COTS VS INTERNAL WILL LEVEL OFF

FURTHER STUDY

As was mentioned earlier, this is the first portion of a study of the overall cost of transition to COTS software engineering tools at Raytheon over a 13 year period. The second part of the study, which will focus on costs, is expected to be completed in 1997.

TOM LYDON, LAURIE FISCHER, KARL GARDNER
RAYTHEON RES, MAILSTOP T3MR8
50 APPLE HILL DRIVE
TEWKSBURY, MA 01876
rtl@swl.msd.ray.com, lpf@swl.msd.ray.com, fkg@swl.msd.ray.com

NASA/GODDARD SOFTWARE ENGINEERING WORKSHOP
Greenbelt, MD - December 4, 1996

TECHNOLOGY EVOLUTION :

COTS Transition at Raytheon 1983-1996

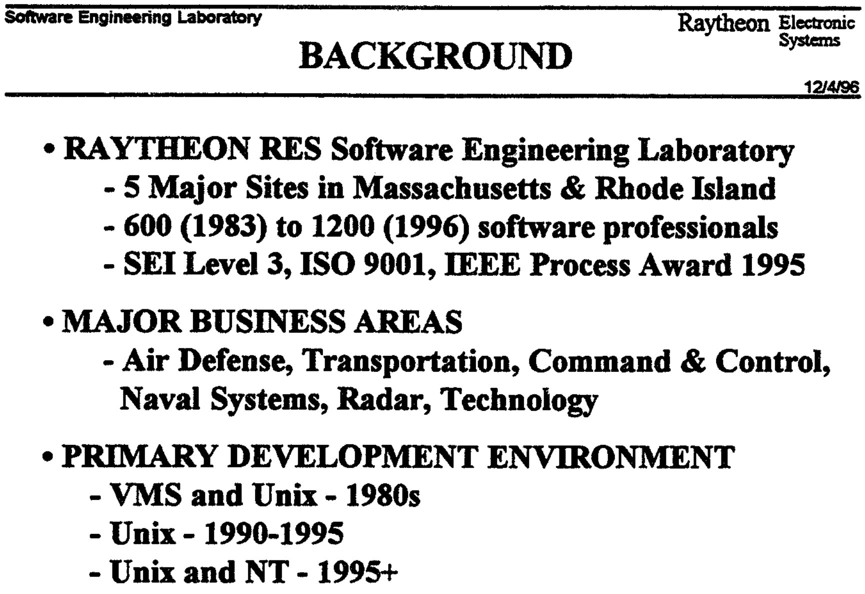
Tom Lydon, Laurie Fischer, Karl Gardner
Raytheon Company

SUMMARY

OVER THE PAST 13 YEARS:

- **Data collected on use of SOFTWARE TOOLS**
- **Increased NUMBER OF TOOLS per Engineer**
- **Shift from INTERNAL tools to COTS tools**
- **Driven by ECONOMICS and CUSTOMER REQTS**
  - **Shift from HW POWER to SW POWER**
  - **COTS are not a PANACEA**
- **Increased PRODUCTIVITY and QUALITY**

WHAT DOES IT MEAN? WHAT'S IN THE FUTURE?



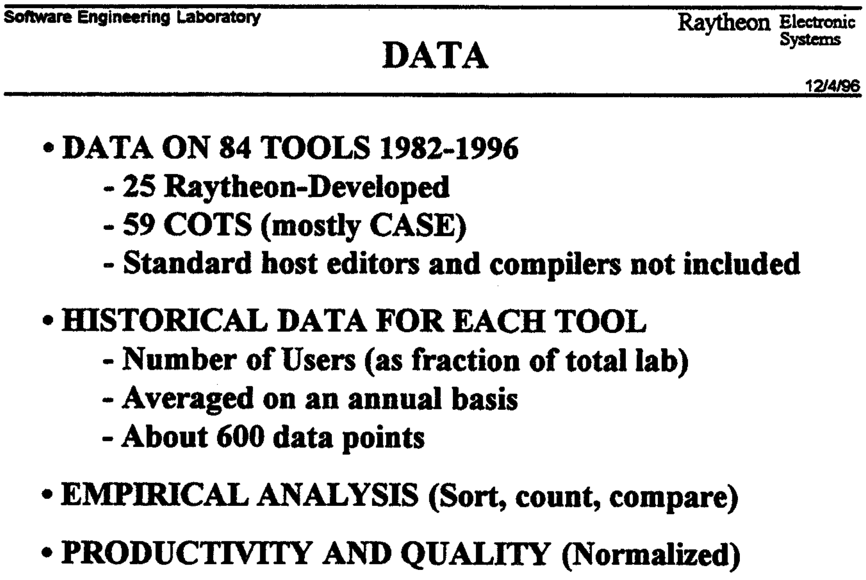
Software Engineering Laboratory

Raytheon Electronic Systems

BACKGROUND

12/4/96

- RAYTHEON RES Software Engineering Laboratory
  - 5 Major Sites in Massachusetts & Rhode Island
  - 600 (1983) to 1200 (1996) software professionals
  - SEI Level 3, ISO 9001, IEEE Process Award 1995
- MAJOR BUSINESS AREAS
  - Air Defense, Transportation, Command & Control, Naval Systems, Radar, Technology
- PRIMARY DEVELOPMENT ENVIRONMENT
  - VMS and Unix - 1980s
  - Unix - 1990-1995
  - Unix and NT - 1995+



Software Engineering Laboratory

Raytheon Electronic Systems

DATA

12/4/96

- DATA ON 84 TOOLS 1982-1996
  - 25 Raytheon-Developed
  - 59 COTS (mostly CASE)
  - Standard host editors and compilers not included
- HISTORICAL DATA FOR EACH TOOL
  - Number of Users (as fraction of total lab)
  - Averaged on an annual basis
  - About 600 data points
- EMPIRICAL ANALYSIS (Sort, count, compare)
- PRODUCTIVITY AND QUALITY (Normalized)

DATA

12/4/96

- **Excerpt from Database**

TYPE	CATEGOR	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
COTS	DOCUM						0.1	0.1	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.01
COTS	MGMT						0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.01		
COTS	CODING						0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.3	0.3
RAY	TEST							0.1	0.1	0.1	0.2	0.2	0.2	0.2	0.2	0.2
COTS	DOCUM							0.1	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
COTS	DESIGN							0.1	0.1	0.2	0.2	0.1	0.2	0.2	0.3	0.3
COTS	REQTS							0.1	0.2	0.3	0.2	0.1	0.1			
COTS	DATA							0.1	0.2	0.4	0.6	0.7	0.7	0.8	0.8	0.8
RAY	TRACE								0.1	0.1	0.1	0.2	0.2	0.2	0.1	0.1
RAY	DEFECT								0.1	0.2	0.3	0.3	0.3	0.4	0.3	0.3
COTS	DOCUM								0.1	0.3	0.6	0.7	0.8	0.8	0.8	0.7
COTS	DESIGN								0.1	0.3	0.3	0.3	0.3	0.3	0.3	0.3
RAY	TRACE									0.1	0.1	0.2	0.2	0.3	0.3	0.2
COTS	DATA									0.1	0.2	0.3	0.2	0.2	0.2	0.2
COTS	GUI									0.1	0.1	0.1	0.1	0.1	0.1	0.2
COTS	MGMT									0.1	0.2	0.3	0.3	0.4	0.5	0.6
COTS	CODING									0.1	0.1	0.1	0.1	0.2	0.2	0.2
COTS	DATA										0.1	0.1	0.1	0.1	0.1	0.1
COTS	DATA										0.1	0.1	0.1	0.1	0.1	0.1
RAY	DEFECT										0.1	0.2	0.3	0.3	0.3	0.2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
COTS	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
RAY	25	0.4	1.2	2.1	3.1	3.6	4.3	4.6	4.7	4.6	4.4	4.4	4.2	4.0	3.6	3.0
COTS	59	0.1	0.4	0.6	1.1	1.3	1.7	2.3	2.8	4.0	5.1	5.9	6.7	7.2	8.1	9.2

Fraction of Laboratory Using Tool

DATA

12/4/96

Category	Raytheon	COTS	TOTAL
CM	3	3	6
CODING	0	9	9
COST	0	4	4
DATA	2	9	11
DB	0	5	5
DEFECT	6	1	7
DESIGN	3	4	7
DOCUM	2	10	12
GIS	0	2	2
GUI	0	4	4
MAINT	3	0	3
MGMT	2	2	4
REQTS	1	4	5
TEST	1	1	2
TRACE	2	1	3
	25	59	84

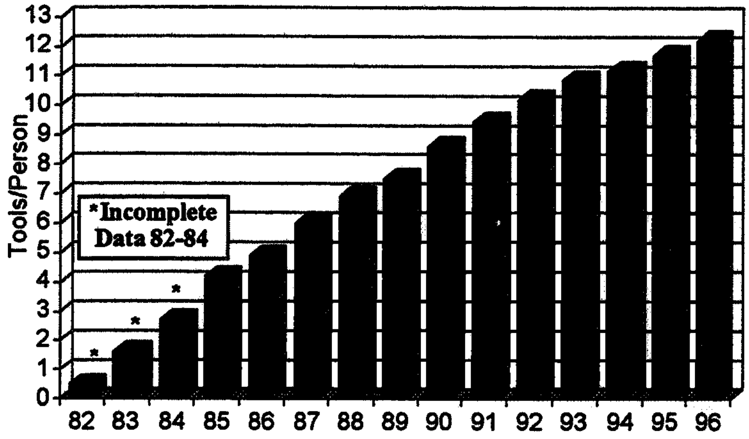
Software Engineering Laboratory

Raytheon Electronic Systems

Overall Tool Use

12/4/96

- **Use of Software Tools has Steadily Increased**



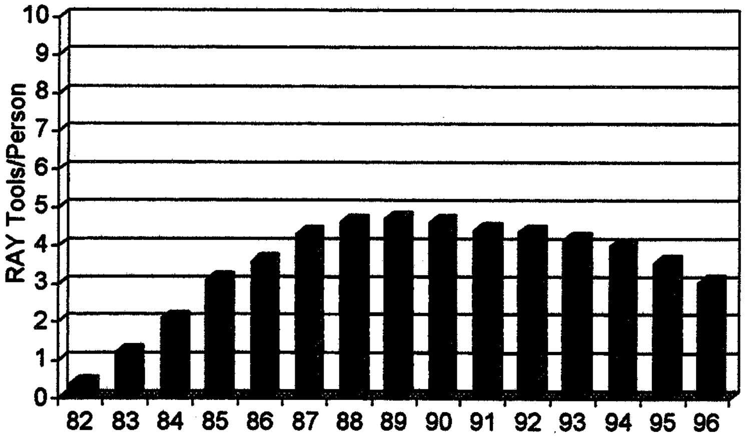
Software Engineering Laboratory

Raytheon Electronic Systems

Raytheon Tools

12/4/96

- **Use of Raytheon-Developed Tools has Dropped Off**

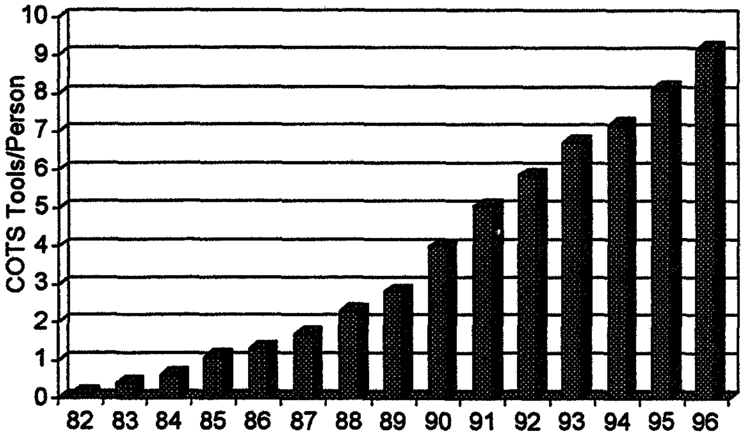


Software Engineering Laboratory Raytheon Electronic Systems

COTS Tools

12/4/96

- **Use of COTS Tools has Increased Dramatically**

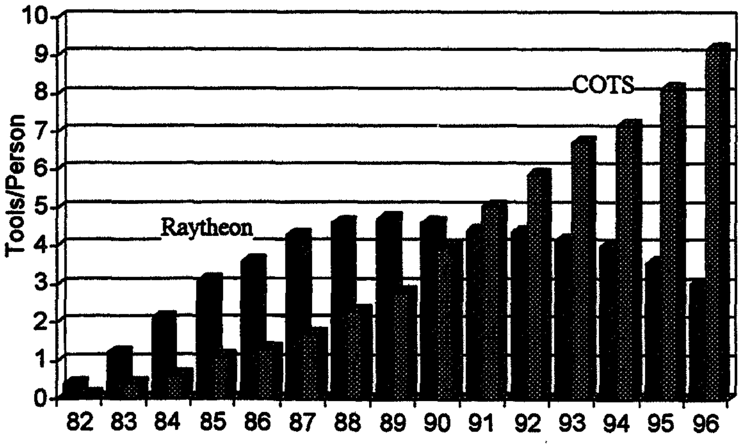


Software Engineering Laboratory Raytheon Electronic Systems

Raytheon vs COTS

12/4/96

- **Counterpoint: Swing from Internal to COTS Tools**

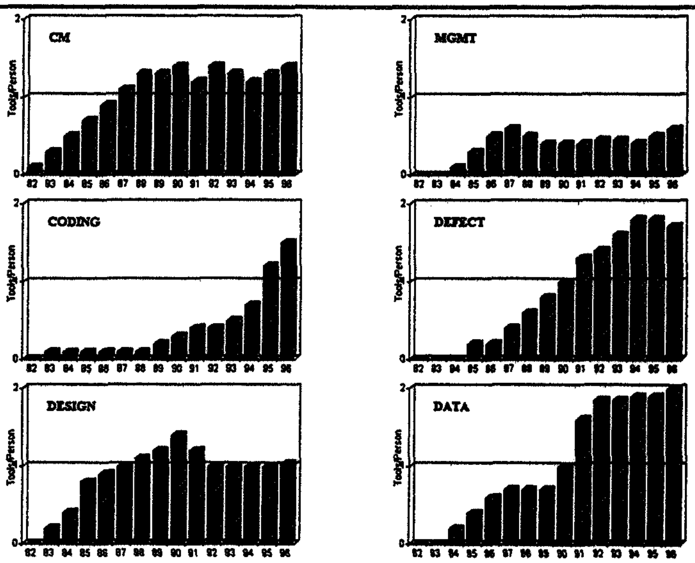


Software Engineering Laboratory

Raytheon Electronic Systems

TOOL CATEGORIES

12/4/96



Software Engineering Laboratory

Raytheon Electronic Systems

PROS & CONS

12/4/96

- **COTS Tools are not a Panacea**

	PROS	CONS
Internal Tools	• Cost is fixed, not proportional to #users • Problem fixes usually faster • New capabilities can be added faster	• More expensive to develop • More manpower required for maint & support • Usually lower quality tools • Slower initial availability
COTS Tools	• Less expensive to develop • Less manpower required for maint & support • Usually better quality tools • Faster initial availability	• Cost increases almost linearly with #users • Problem fixes usually slower • New capabilities usually not added quickly

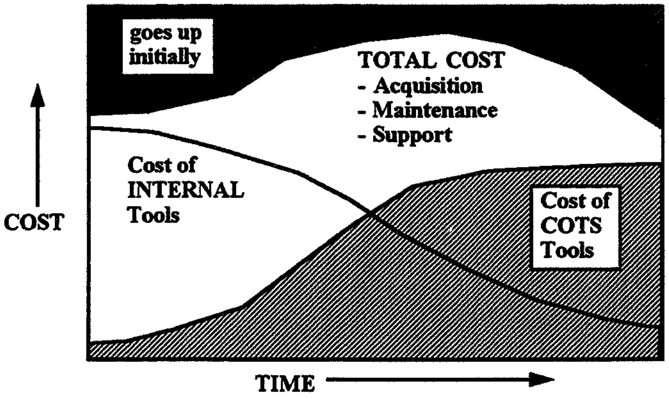
Software Engineering Laboratory

Raytheon Electronic Systems

COTS COST

12/4/96

- **During Transition, costs initially go UP**



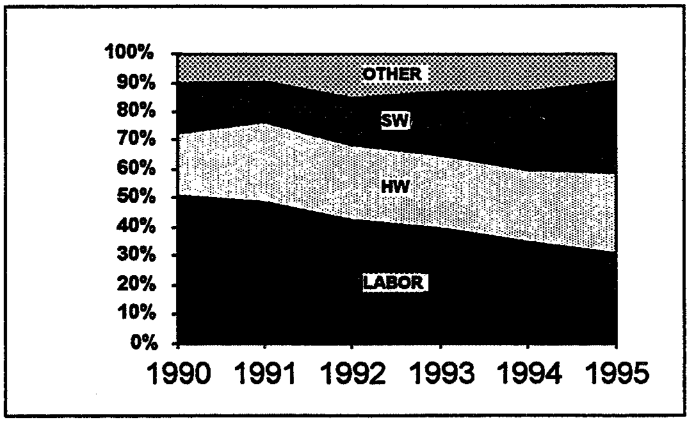
Software Engineering Laboratory

Raytheon Electronic Systems

SUPPORT COST

12/4/96

- **Costs Shifting Towards Software (esp. Maintenance)**

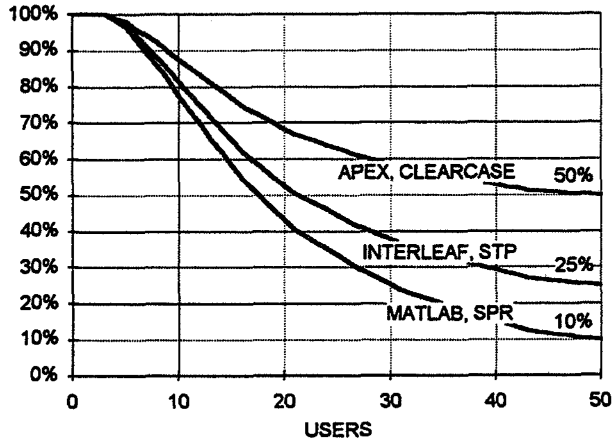


Software Engineering Laboratory Raytheon Electronic Systems

LICENSE REQUIREMENTS

12/4/96

- **For large networks, fewer licenses required per user**



Software Engineering Laboratory Raytheon Electronic Systems

STANDARDIZATION

12/4/96

RAYTHEON USES A STANDARD SOFTWARE LIST

- **PROGRAMS & DEPTS HAVE OWN PREFERENCES**
- **STANDARDS PROVIDED "FREE" BY COST CENTER**
- **FINANCIAL INCENTIVE TO USE STANDARD**
- **MINIMIZES NUMBER OF SUPPLIERS**
- **MAXIMIZES LEVERAGE WITH EACH SUPPLIER**
- **SOME NON-STANDARD TOOLS STILL REQUIRED**

- **1994 - 18 STANDARD TOOLS (1 Division)**
- **1995 - 26 STANDARD TOOLS (Merged Divisions)**
- **1996 - 22 STANDARD TOOLS (Budget Constraints)**

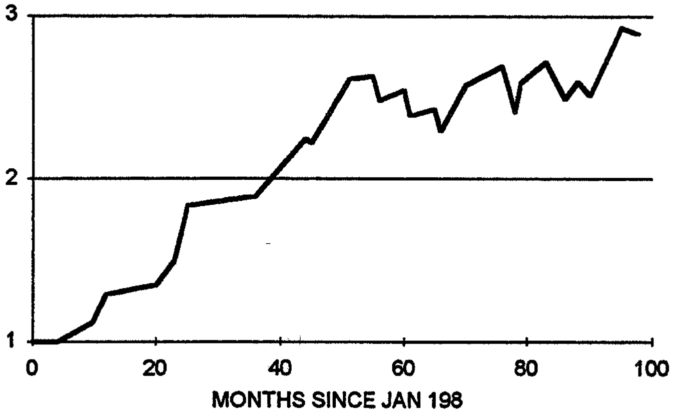
Software Engineering Laboratory

Raytheon Electronic Systems

PRODUCTIVITY

12/4/96

- **Tools help improve productivity**



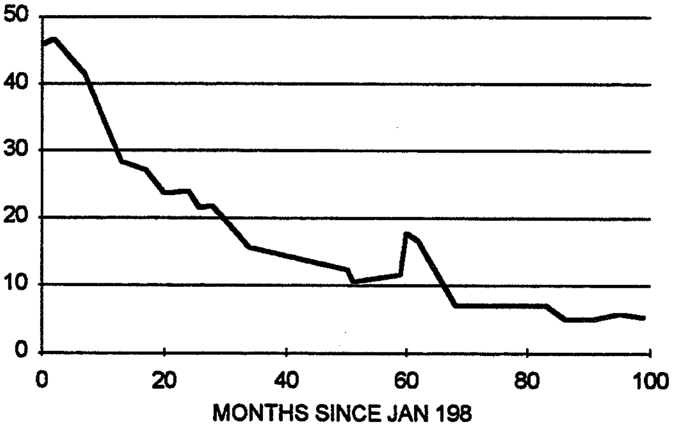
Software Engineering Laboratory

Raytheon Electronic Systems

DEFECTS/REWORK

12/4/96

- **Tools help reduce defects & rework**



Software Engineering Laboratory

Raytheon Electronic Systems

OTHER METRICS

12/4/96

OTHER OBSERVATIONS IN THE DATA:

- **Total UNIQUE TOOLS in use increased from 4* to 62**
  - **- 14 of 62 are almost dead, leaving 48 truly active**
- **Number of DEATHS = 22, Average ~2 per year**
  - **- Tools are often slow to actually die**
- **Average LIFESPAN of 22 Dead Tools = 4.4 years**
- **COMMON Tools (reaching >25% of Lab) = 39%**
- **NICHE Tools (never reaching 25% of Lab) = 43%**
  - **- Remaining 18% too soon to tell**

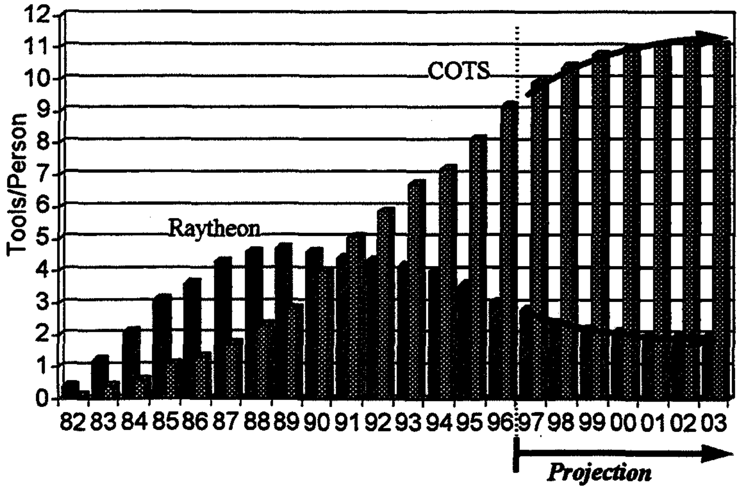
Software Engineering Laboratory

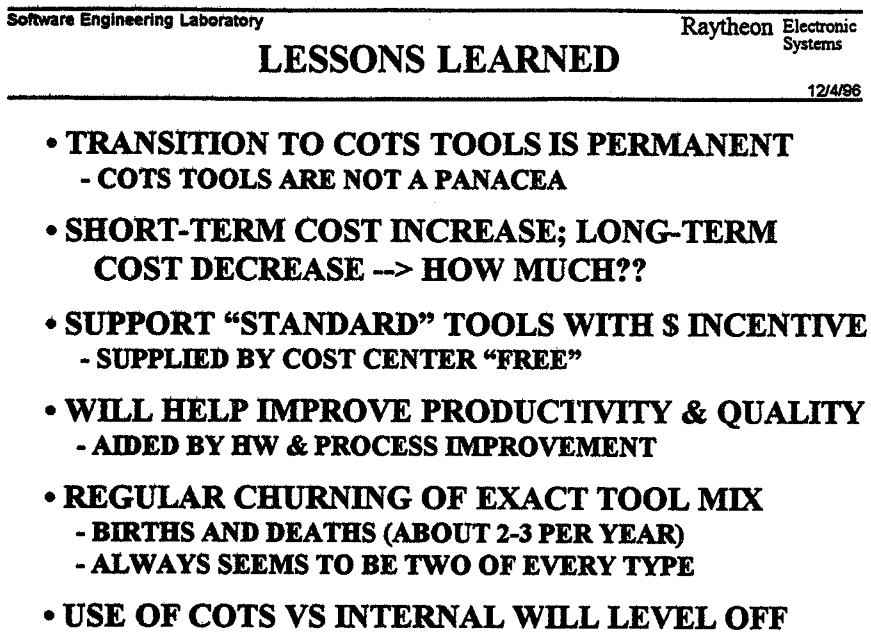
Raytheon Electronic Systems

FUTURE?

12/4/96

- **Tool Saturation, Stabilized Levels**





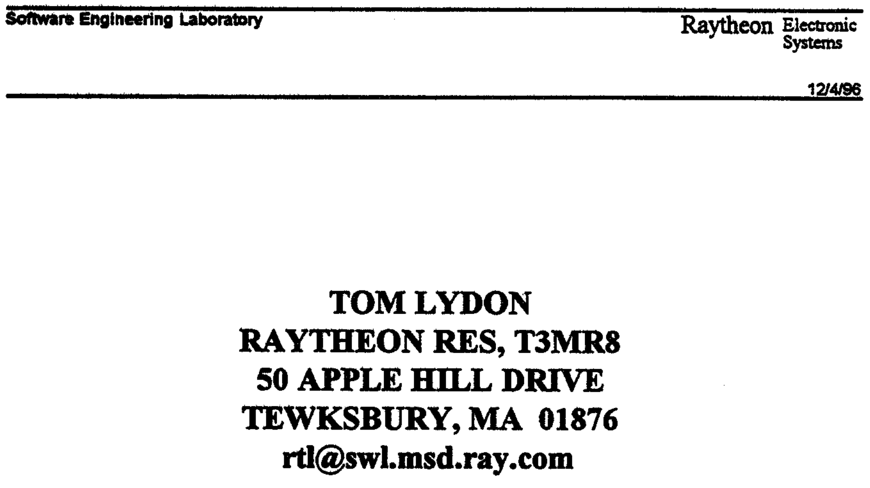
Software Engineering Laboratory

Raytheon Electronic Systems

LESSONS LEARNED

12/4/96

- TRANSITION TO COTS TOOLS IS PERMANENT
  - COTS TOOLS ARE NOT A PANACEA
- SHORT-TERM COST INCREASE; LONG-TERM COST DECREASE --> HOW MUCH??
- SUPPORT "STANDARD" TOOLS WITH $ INCENTIVE
  - SUPPLIED BY COST CENTER "FREE"
- WILL HELP IMPROVE PRODUCTIVITY & QUALITY
  - AIDED BY HW & PROCESS IMPROVEMENT
- REGULAR CHURNING OF EXACT TOOL MIX
  - BIRTHS AND DEATHS (ABOUT 2-3 PER YEAR)
  - ALWAYS SEEMS TO BE TWO OF EVERY TYPE
- USE OF COTS VS INTERNAL WILL LEVEL OFF



Software Engineering Laboratory

Raytheon Electronic Systems

12/4/96

TOM LYDON
RAYTHEON RES, T3MR8
50 APPLE HILL DRIVE
TEWKSBURY, MA 01876
rtl@swl.msd.ray.com

[illegible]
[illegible]

Session 4: Reliability

Identification of Failure–Prone Modules in Two Software System Releases

5/0-61
415654
360721
p 12

Niclas Ohlsson and Claes Wohlin
Dept. of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
E–mail: (nicoh, clawo)@ida.liu.se

1 Introduction

This paper presents a case study of fault and failure data from two consecutive releases of a large telecommunication system. In this context it is important to have clear interpretations of errors, faults and failures. Thus, we would like to make the following distinction between them. Errors are made by humans, which may result in faults in the software. The faults may manifest themselves as failures during operation. Thus, faults can be interpreted as defects in the software and failures are the actual malfunction in an operational environment. In this paper we have used *fault–prone modules* to denote the modules that account for the highest number of faults disclosed during testing, while *failure–prone modules* is used to denote the modules accounting for the highest number of faults disclosed during the first office application and in operation. The general objective of the study is to investigate methods of identifying failure–prone software modules. Furthermore, the goal is to use the knowledge acquired to improve the software development process in order to improve software quality in the future.

Some early results using parametric statistics have been reported in (Ohlsson and Alberg, 1996). The models have since been refined and analysed with non–parametric statistics (Ohlsson et al., 1996). Identification of fault–prone modules has also been addressed by other researchers (Khoshgoftaar and Kalaichelvan, 1995) and (Munson and Khoshgoftaar, 1992). Few, if any, studies have exploited the opportunities to identify not only fault–prone modules, but also failure–prone modules which are the main concern of the user. There is also a general lack of studies investigating whether identification of fault–prone modules means that we actually also identify failure–prone modules.

Another important issue is to establish when in the development phase we are able to identify modules which will be failure–prone in the operational phase. This paper investigates three different times for prediction: history (previous release), the design phase and the test phase. One important consideration is to address whether or not fault–prone modules during testing are failure–prone during operation. If fault–prone does not imply failure–prone, then we may have to improve the test methods.

The paper is organized as follows. In Section 2, an overview of the study is presented. Section 3 discusses identification of failure–prone modules based on experience from a previous release, and Section 4 presents results using prediction models based on design measures. In Section 5, results concerning identification of failure–prone modules based on test data are presented. Finally, some conclusions are given in Section 6.

2 Overview of study

This paper is part of a long–term empirical study conducted at Ericsson Telecom AB with the objective of studying how identification of fault and failure–prone modules can be used to achieve cost–effective quality improvement. In release n of the system 130 modules have been analysed and in release n+1 232 modules have been investigated. Fault and failure data have been collected from functional testing, system testing, first office application (i.e. the first 26 weeks and a number of site tests) and operation. It was possible to trace 69 modules developed for release n that were modified in release n+1. Release n+1 is a major system revision. Data is currently being collected for release n+2. The modules are of the size of 1000 to 6000 lines of code each.

Promising results concerning identification of fault–prone modules have been presented elsewhere, i.e. design measures were used to identify fault–prone modules (Ohlsson et al., 1996) and (Ohlsson and Alberg, 1996). The objective here is to study the identification of failure–prone modules based on fault and failure data as well as from design measures. In this paper we have used one failure as threshold for the dependent variable, i.e. modules with one or more failures are classified as failure–prone. The underlying analysis of design measures is based on ordinal analysis, as it allows for changing the threshold with regards to what are viewed as being fault– and failure–prone modules (Ohlsson et al., 1996). Actual threshold–values are not recommendations; thresholds should be determined in individual projects on the basis of, for example, the level of criticality of the system and market requirements. The primary objective of the thresholds as presented in this paper is to illustrate the outcome when applying the methods for identification of failure–prone modules.

The predictability of the different models is viewed in Contigency tables and the kappa coefficients are calculated to measure the agreement in classification of the modules (Siegel and Castellan, 1988). The kappa coefficient is the ratio of the proportion of times that the classifications is correct to the maximum proportion of times that the classifications could be correct. If the classifications completely agree, then kappa=1; whereas if there is no agreement between the classifications, then kappa=0. Kappa will assume -1 if there is a perfect missclassification.

The study is divided into three parts:

1. Identification of failure–prone modules using data from a previous release

   This part is aimed at investigating whether the information from release n concerning fault– and failure–prone modules is a good predictor of failure–prone modules in release n+1. More than 90 percent of the modules in release n had one or more faults. Therefore, it is infeasible to use one fault as a threshold. Thus, when fault–prone modules from release n is used to predict failure–prone modules in release n+1, a threshold of five faults is used for the independent variable as an indication of potential failure–prone modules. When failure–prone modules in release n are used as the independent variable, one failure is used as threshold.

2. Identification of failure–prone modules using design measures

   The initial objective was to build prediction models in release n for identification of failure–prone modules based on design measures, which then should be validated with data from release n+1. Due to variation in quality between the two releases this was not possible. Instead design metrics were only evaluated within release n+1. Only the best design measure is reported here, as the main objective is to investigate different opportunities to identify failure–

prone modules rather than evaluate which measures are the best predictors. To the best of our knowledge there exists no empirical evidence that complexity values higher than a specific threshold would indicate either fault– or failure–prone modules. However, there are results suggesting relative stable distribution in line with the Pareto principle (Ohlsson et al., 1996). Therefore, the threshold is based on the percentage of failure–prone modules in release n+1. That is, 29 percent of the modules in n+1 had one or more failures. Hence, this percentage value is used as a threshold for the design measures.

3. Identification of failure–prone modules from fault–prone modules

   The objective of this part is to investigate whether the fault–prone modules identified in release n and n+1 are good indicators of failure–prone modules in the two releases. This means that fault data from testing is used to predict failure–proneness during operation. The rationale for selecting thresholds is the same as in part 1.

To summarize, the main difference is when prediction can be made. The three parts imply three different points of time in a project, namely: project start (part 1), design phase (part 2), and testing phase (part 3). It is important to remember that the sooner we are able to identify modules which are likely to be failure-prone, the sooner we can take appropriate measures to deal with them. For example, we can allocate the best people, intensify inspections or take other special improvement measures.

3 Failure–prone modules from history

For software systems, it is normal practice that a system is regularly upgraded and released in new versions. This implies that some parts of the system are the same in different releases. This information can be used to apply experience from one release to the next release or following releases. In this empirical study, the hypothesis is that fault– or failure-prone modules in release n are likely candidates for being failure–prone in release n+1. It was possible to trace 69 modules developed for release n that were modified in release n+1. The data from the historical analysis is shown in Table 1. It should be noted that only four modules were failure–prone in release n, see analysis A, while 18 modules were failure–prone in release n+1.

To evaluate the goodness of the predictions, the prediction errors must be considered. This includes two different types of errors: failing to identify failure–prone modules and identification of modules as failure–prone when they are not. These are hereafter referred to as errors of type I and II respectively. It should be noted that a correct identification means actually pin–pointing a certain module correctly.

To evaluate the goodness of the predictions, the prediction errors must be considered. This includes two different types of errors: failing to identify failure–prone modules and identification of modules as failure–prone when they are not. These are hereafter referred to as errors of type I and II respectively. It should be noted that a correct identification means actually pin–pointing a certain module correctly.

TABLE 1. Failures identified in release n+1 based on release n.

	Analysis A[a]		Analysis B[b]		Analysis C[c]	
	Threshold=1 Failure(n)		Threshold=5 Fault(n)		Threshold=5 Fault+ Failure(n)	
Actual	F	Not F	F	Not F	F	Not F
Failure-prone(n+1) (18 observation)	4	14	14	4	15	3
Not Failure-prone(n+1) (51 observations)	0	51	28	23	28	23
Total observations	4	65	42	27	43	26
Misclassifications of type I and II	78% (14/18)	0% (0/51)	28% (4/18)	55% (28/51)	17% (3/18)	55% (28/51)
Overall misclassifications	20% (14/69)		46% (32/69)		45% (31/69)	

a. Kappa 0.30
b. Kappa 0.16
c. Kappa 0.32

Analysis A in Table 1 illustrates that even though the type I error is as high as 78%, there is no type II error. This means that the modules that are failure–prone in release n are all failure–prone in release n+1. Possible explanations for this are the actual type of failure and late erroneous fault correction in test.

For analyses B and C, we have used five faults as a threshold for the independent variable. It has earlier been suggested (Khoshgoftaar and Kalaichelvan, 1995) that this should be used as threshold for fault–prone modules. The threshold could therefore indicate failure–proneness. Using one fault is not reasonable since this would identify 63 modules as being failure–prone. Even with a threshold of five faults in analysis B as many as 61 percent (42/69) of the modules are identified in release n as failure–prone. However, only 78 percent (14/18) of all the failure–prone modules in release n+1 are identified. Therefore, fault–prone modules in release n are poor predictors of failure–prone modules in n+1. This is also true for analysis C.

Another possible alternative would be to select a threshold based on the percentage of failure–prone modules in release n+1, i.e. assuming that this proportion of fault– and failure–prone modules will be stable over later releases. The number of potential failure–prone modules would be more realistic using 26 percent (18/69) as a threshold. However, only 28 percent of the failure–prone modules would be identified. This also holds for analysis C. Therefore, the two models in analyses B and C are not applicable.

4 Failure–prone modules from design measures

Earlier studies (Ohlsson et al., 1996) have indicated that models built on design metrics are worthwhile when the total number of faults and failures are considered as the dependent variable. Thus, it is reasonable to try this approach for failure–prone modules. In this study, fourteen different design measures are used to build prediction models for release n+1. Spearman's correlation coefficient (Siegel and Castellan, 1988) was used for a first analysis. All potential variables have low correlation values (below 0.35). There was, however, a rather low correlation among some of the variables, hence it could be possible to improve the model by combining the variables into more complex models. Multiplicative aspects of the potential variables will be investigated in later studies. In this particular case, the best design measure predictor was IS, which is the number of input–signals for a module in the design. The result was later compared with lines of code, which was found to be doing even worse.

It has been suggested that prediction models should first be developed for one release, validated in the succeeding release, and then applied in the third release. However, the quality of the two releases varied widely, and it was therefore not possible to do so in this study. From a modelling point of view, the number of failure–prone modules in release n was too few. Instead, the explanatory ability of design metrics was evaluated by building the best possible model based on data in release n+1. The results shown in Table 2 are based on a threshold of one failure, which corresponds to 29 percent of the modules.

TABLE 2. Failures identified in release n+1 based on IS(n+1).

	Analysis[a]	
	IS(n+1)	
Actual	F	Not F
Failure-prone(n+1) (67 observation)	28	39
Not Failure-prone(n+1) (165 observations)	39	126
Total observations	67	165
Misclassifications	58% (39/67)	24% (39/165)
Overall misclassifications	34% (78/232)	

[a] Kappa 0.18

From Table 2, it can be seen that the explanatory ability is unsatisfactory, i.e. the misclassification is too high, including a large proportion of both type I and II errors. This, in combination with the fact that the quality of the two releases differed, suggests that more complete models should be investigated, for example including verification effort and quality.

5 Failure–prone modules from fault–prone modules

The data from the testing phase can be used for both releases to predict the failure–prone modules. The problem with choosing relevant thresholds, discussed in respect to part 1, is relevant for this part, too. The results of the analyses are shown in Table 3, using a threshold of five faults for the independent variable.

TABLE 3. Failures identified based on faults disclosed during testing of release n and n+1 respectively.

	Analysis n[a]			Analysis n+1[b]	
	Fault(n)			Fault(n+1)	
Actual	F	Not F	Actual	F	Not F
Failure-prone(n) (13 observation)	5	8	Failure-prone(n+1) (67 observation)	47	20
Not Failure-prone(n) (117 observations)	77	40	Not Failure-prone(n+1) (165 observations)	102	63
Total observations	82	48	Total observations	147	83
Misclassifications	62% (8/13)	66% (77/117)	Misclassifications	30% (20/67)	62% (102/165)
Overall misclassifications	65% (85/130)		Overall misclassifications	53% (122/232)	

[a] Kappa -0.08
[b] Kappa 0.06

The misclassification is also too high in this analysis. This means that modules that are fault–prone during testing are not failure–prone. A possible explanation is that other types of defects are discovered in operation, such as performance problems, that are difficult to test. This explanation is supported by experienced developers from Ericsson. This could also explain the result in part 1. A possible explanation of the fact that failure–prone modules in n are failure–prone in n+1 could be that modules which are critical from a capacity perspective in release n, will remain so in release n+1. The results indicate the need for a better understanding of the types of defects that result in failures and the types of the failures themselves. The results also stress the need to identify factors causing the defects which result in failures. Increased understanding is essential for quality improvement.

6 Conclusions

In this paper we have investigated the opportunity to predict failure–prone modules based on fault and failure data from two succeeding releases, design metrics, as well as test data. The study revealed that failure–prone modules in release n are failure–prone in n+1. Other suggested independent variables are poor predictors of failure–proneness. However, this is not the same as saying that they do not explain any of the variation. It only means that on their own they are poor explanatory factors. Instead, the study suggests that methods that combine these different independent variables are needed.

In this study, we have addressed two consecutive releases of a software system. This is an important aspect as in most cases it is not possible to both build, validate and use a prediction model within one release. It is, thus, important to investigate how to build models in one release, validate the model in the next release and then use the model in the third release. The transferability of a model between a software system's releases is crucial to success in the mission of identifying failure–prone modules prior to the operational phase.

A major problem with predictions is that failures are dynamic, hence it may be difficult to identify failure–prone modules using static measures. This is an issue which has to be further studied. One potential solution would be to take the use of modules into account when predicting failure–proneness. This would allow for capturing the dynamic aspects of usage in the independent variable.

Another important issue which has been addressed here is the point of time when we are able to identify failure–prone modules. To improve the usefulness of the predictions, they should preferably be done at an early stage. In this study, we have focused on data from the previous release, the design and the test phase. The knowledge from the previous release is important in identifying failure–prone modules, but this is not a feasible approach for new modules. Thus, it is very important to find early indicators of failure–proneness, since this is the only way to enable us to address the problem within the same release.

Models which identify failure–prone modules are important not only in enabling prediction during the operational phase, but also as a planning and control tool during development. Managers may use these models to improve the resource allocation for design, both in terms of effort and experience. Furthermore, knowing which modules are most likely to be failure–prone in operation suggest that the modules will be tested and inspected differently. Therefore more attributes need to be considered and incorporated in the models, for example verification effort and quality, in line with Fenton et al. (Fenton et al., 1995), to explain the variation and to be able to apply the models in subsequent releases.

Future work should not only aim at building these more complete models, but also aim at investigating additative and multiplicative aspects of design measures and measures from different phases, in order to gain more knowledge about how such a component fits into a more complete model. The results in this study also suggest that prediction models that are only based on test data will have limited applicability in real projects aiming at addressing operational issues.

Acknowledgement
The authors would like to thank Ericsson Telecom AB for supporting this empirical study.

References

Fenton, N. E., Neil, M., and Ostrolenk, G. (1995). Metrics and models for predicting software defects. Technical Report CSR/10/02, Centre for Software Reliability, City University, London, UK.

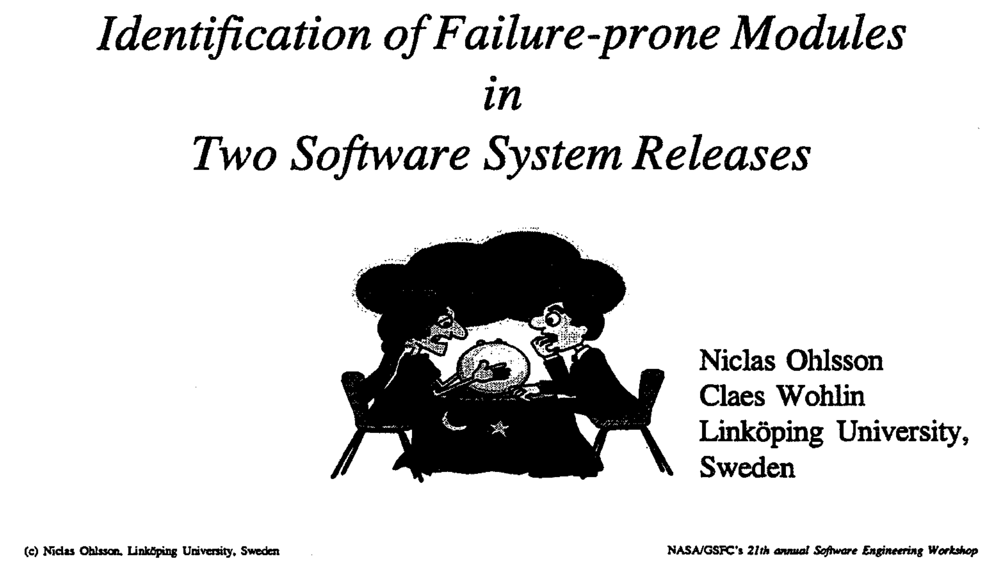
Khoshgoftaar, T. M. and Kalaichelvan, K. S. (1995). Detection of fault-prone programs modules in a very large telecommunication system. In Proceedings of *The Sixth International Symposium on Software Reliability Engineering*, pages 24–33, Toulouse, France.

Munson, J. C. and Khoshgoftaar, T. M. (1992). The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433.

Ohlsson, N. and Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *To appear in IEEE Transactions on Software Engineering.*

Ohlsson, N., Helander, M., and Wohlin, C. (1996). Quality improvement by identification of fault-prone modules using software design metrics. In Proceedings of *Sixth International Conference of Software Quality*, pp. 1-13, Ottawa, Canada.

Siegel, S. and Castellan, N. J. J. (1988). *Nonparametrics Statistics for the Behavioral Sciences.* McGraw-Hill, second edition.

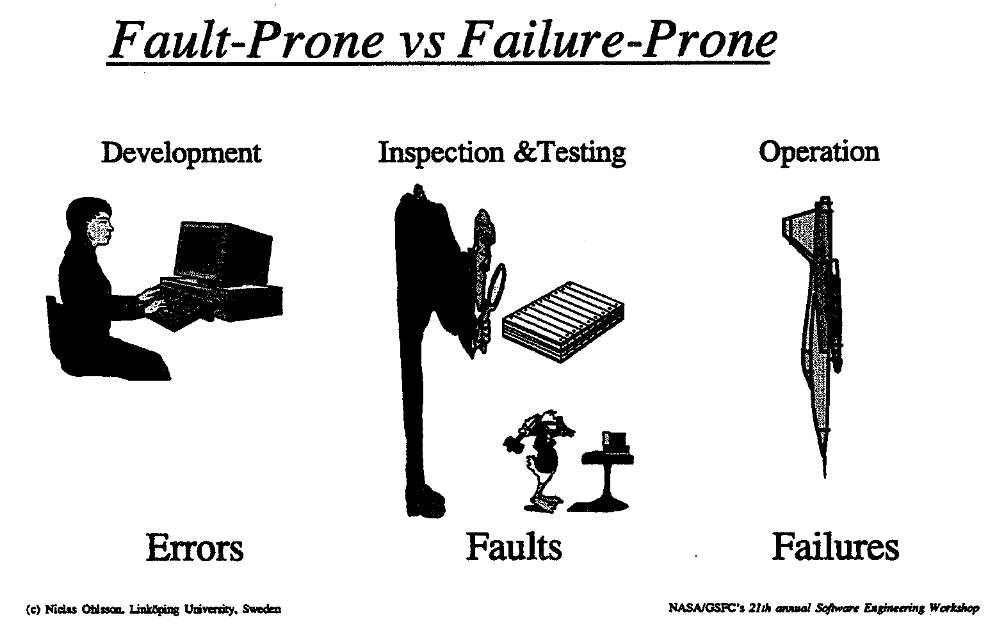


Identification of Failure-prone Modules in Two Software System Releases

Niclas Ohlsson
Claes Wohlin
Linköping University,
Sweden

(c) Niclas Ohlsson, Linköping University, Sweden

NASA/GSFC's *21th annual Software Engineering Workshop*



Fault-Prone vs Failure-Prone

Development	Inspection &Testing	Operation
Errors	Faults	Failures

(c) Niclas Ohlsson, Linköping University, Sweden

NASA/GSFC's *21th annual Software Engineering Workshop*

Three Parts

- Identification of failure-prone modules using data from a previous release
- Identification of failure-prone modules using design metrics
- Identification of failure-prone modules from fault-prone modules

Discriminant Analysis

- Dependent Variable

  Failure-prone = Fault disclosed in operation > 0

- Independent Variable

  Specifying Threshold

- Type I and Type II error

Models Based on Previous Release

- ∗ Three analyses

  Failure(n), Fault(n), and Failure(n)+Fault(n)

  ⇨ Failure-prone(n+1) ?

- ∗ Failure-prone(n) ⇨ Failure-prone(n+1)

(c) Niclas Ohlsson, Linköping University, Sweden

NASA/GSFC's *21th annual Software Engineering Workshop*

Models Based on Design Metrics

- ∗ 14 different metrics
- ∗ Low correlation
- ∗ Methods to combine ordinal metrics

(c) Niclas Ohlsson, Linköping University, Sweden

NASA/GSFC's *21th annual Software Engineering Workshop*

Models Based on Fault-Proneness

* Two analyses

Fault(n) ⇒ Failure-prone(n)
Fault(n+1) ⇒ Failure-prone(n+1)

* Very high misclassification

Summary

* Failure-prone(n) ⇒ Failure-prone(n+1)

* Design and testing data: low explanatory ability

* Lack dynamic aspects

* More complete models

Combining measures of one attribute
Combining attributes of product, process, and resources (BBN)

S11-61
415655
360724
p. 14

Proceedings of 21st Annual Software Engineering Workshop
NASA/Goddard Space Flight Centre, December 4-5, 1996

Predicting Software Quality using Bayesian Belief Networks

Martin Neil & Norman Fenton
Centre for Software Reliability
City University
Northampton Square
London EC1V OHB, UK

Abstract

In the absence of an agreed measure of software quality the *density of defects* has been a very commonly used surrogate measure. As a result there have been numerous attempts to build models for predicting the number of residual software defects. Typically, the key variables in these models are either size and complexity metrics or measures arising from testing information. There are, however, serious statistical and theoretical difficulties with these approaches. Using Bayesian Belief Networks we can overcome some of the more serious problems by taking account of all the diverse factors implicit in defect prevention, detection and complexity.

1. Background

For the last 20 years the software engineering community has spent much effort in trying to answer the question, "Can we predict the quality of our software *before* we use it?". There are literally scores of papers, articles and reports advocating statistical models, metrics and solutions which purport to answer this question. Generally, efforts have tended to concentrate solely on one of the following three problem perspectives:

a) Predicting the number of defects in the system using software size and complexity metrics

The earliest study of the relationship between defects and complexity appears to have been [Akiyama,1971] which was based on a system developed at Fujitsu, Japan. It is typical of many regression based "data fitting" models which became common-place in the literature (such as [Ferdinand 1974], [Lipow 1982], [Gaffney 1984], [Basili and Perricone 1984], [Shen 1985], [Compton and Withrow 1990], [Moller and Paulish 1993]). The study showed that linear models of some simple metrics provide reasonable estimates for the total number of defects *d* (the dependent variable) which is defined as the sum of the defects found during testing and the defects found during two months after release. Although there is no convincing evidence to show that any of the hundreds of published complexity metrics are good predictors of defect density, there *is* a growing body of evidence that some of these metrics may be useful in outlier analysis (especially when grouped together) [Bache and Bazzana 1993]—they can be used to predict which of a set of modules is likely to be especially defect-prone.

b) Inferring the number of defects from testing information

Some of the most promising local models for predicting residual defects involve very careful collection of data about defects discovered during early inspection and testing phases. A notable example of this is reported by the IBM NASA Space shuttle team [Keller 1992]. Another class of testing metrics that appears to be quite promising for predicting defects is the class of so called *test coverage measures*. [Fenton and Pfleeger 1996]. For a given strategy and a given set of test cases we can ask what proportion of coverage has been achieved. The resulting metric is defined as the Test Effectiveness Ratio (TER) with respect to that strategy. Clearly we would expect defect rate to decrease as the values of these metrics increases. [Veevers and Marshall 1994] report on some defect and reliability prediction models using these metrics which give quite promising results.

c) Assessing the impact of design or process maturity on defect counts.

There are many experts who argue that the quality of the development process is the best predictor of product quality. The simplest metric of process quality is the 5-level ordinal scale SEI Capability Maturity Model ranking. Despite its widespread popularity, there is no convincing evidence to show that higher maturity companies generally deliver products with lower residual defect rate than lower maturity companies. Nevertheless, this seems to be a widely held assumption and is therefore important in explaining and predicting defects.

2. The need to take account of diverse factors

Despite the many efforts described above there appears to have been little overall improvement in the accuracy of the predictions made using these models (if predictions are *formally* made at all) or indeed whether the models make sense. Broadly speaking there are a number of serious statistical and theoretical difficulties that have caused these software quality prediction problems ([Neil 1992] provides explicit criticisms of many of the models). To avoid these problems we need to take account of all the diverse factors implicit in defect prevention, detection and complexity.

Perhaps the most critical issue in any scientific endeavour is agreement on the constituent elements or variables of the problem under study. Models are developed to represent the salient features of the problem in a systemic fashion. This is as much the case in physical sciences as social sciences. For instance, in macro-economic prediction we could not predict the behaviour of an economy without an integrated, complex, model of all of the known, pertinent variables. Choosing to ignore or forgetting to include key variables such as *savings rate* or *productivity* would make the whole exercise invalid and meaningless. Yet this is the position that many software practitioners are in - they are being asked to accept simplistic models which are missing key variables that are already *known* to be enormously important. Predicting the number of defects discovered based on lines of code alone is as much use as predicting a person's IQ from a knowledge of their shoe size.

Our view is that the isolated pursuit of these single issue perspectives on the quality problem are, in the longer-term, fruitless. The solution to many of the difficulties presented above is to develop prediction models that *unify* the diverse software quality prediction models. This unification will help produce new systematic models that better represent the complex relationships inherent in software engineering. Only when such unified models are developed will statistical experimentation and then practical use be warranted.

As well as facing up to the complexity inherent in software engineering we must also recognise that modelling the actions of the designer and manager are crucial if we are to predict the quality of the final product. Again and again experience dictates that it is good managers and designers that determine the difference between failure and success. However researchers have tended to ignore the issue of human intervention even though we know it is *the* key variable in software design. A consequence of this is that subjectivity and uncertainty is all pervasive in software development. Project managers make decisions about quality and cost using best guesses; it seems to us that will always be the case and the best that researchers can do is a) recognise the fact and b) improve the 'guessing' process.

The results of inaccurate modelling and inference is perhaps most evident in the debate that surrounds the 'Is Bigger Better?' dilemma. This is the phenomenon that larger modules have lower defect densities [Basili and Perricone 1984] and [Shen 1985]. [Moller and Paulish 1993] provide further evidence, and also examined the effect of modifications and reuse on defect density. Similar experiences are reported by [Hatton 1993, 1994]. Basili and Perricone argued that this may be explained by the fact that there are a large number of interface defects distributed evenly across modules, and that larger modules tend to be developed more carefully. Others have mentioned the possible effects of testing.

The notion that larger modules have lower defect density is surprising because it questions the whole edifice of problem and design decomposition so central to software engineering. It suggests that building bigger modules will result in less defects overall. To act on these results would mean throwing away much of what is being advocated in structured, object-oriented and formal design - 'Why should we apply decomposition when it doesn't improve quality?'. Post-hoc explanations cannot easily dismiss the uncomfortable significance of this result.

3. Bayesian Belief Networks (BBNs)

Achieving the above modelling challenges appear onerous when one considers the tools previously available to researchers and practitioners. They have had to rely on the power of classical statistical analysis tools, such as regression, discriminant analysis and correlation. Classical methods demand simple linear structures and a wealth of data so often missing in software engineering. These methods have severely restricted the scale of problems that could be tackled. However, a relatively new but rapidly emerging technology has provided an elegant solution enabling us to push back the boundary of the problems that can be attacked: *Bayesian Belief Networks* (BBNs) [Pearl, 1988].

A BBN is a graphical network that represents probabilistic relationships among variables. BBNs enable reasoning under uncertainty and combine the advantages of an intuitive visual representation with a sound mathematical basis in Bayesian probability. With BBNs, it is possible to articulate expert beliefs about the dependencies between different variables and to propagate consistently the impact of evidence on the probabilities of uncertain outcomes, such as 'future system reliability'. BBNs allow an injection of scientific rigour when the probability distributions associated with individual nodes are simply 'expert opinions'. A BBN will derive all the implications of the beliefs that are input to it; some of these will be facts that can be checked against the project observations, or simply against the experience of the decision makers themselves. There are many advantages of using BBNs, the most important being the ability to represent and manipulate complex models that might never be implemented using conventional methods. Because BBNs have a rigorous, mathematical meaning there are software tools that can interpret them and perform the complex calculations needed in their use. The specific tool used here is *Hugin Explorer* [Hugin 1996], which provides a graphical front end for inputting the BBNs in addition to a computational engine for the Bayesian analysis.

4. The Defect Density BBN

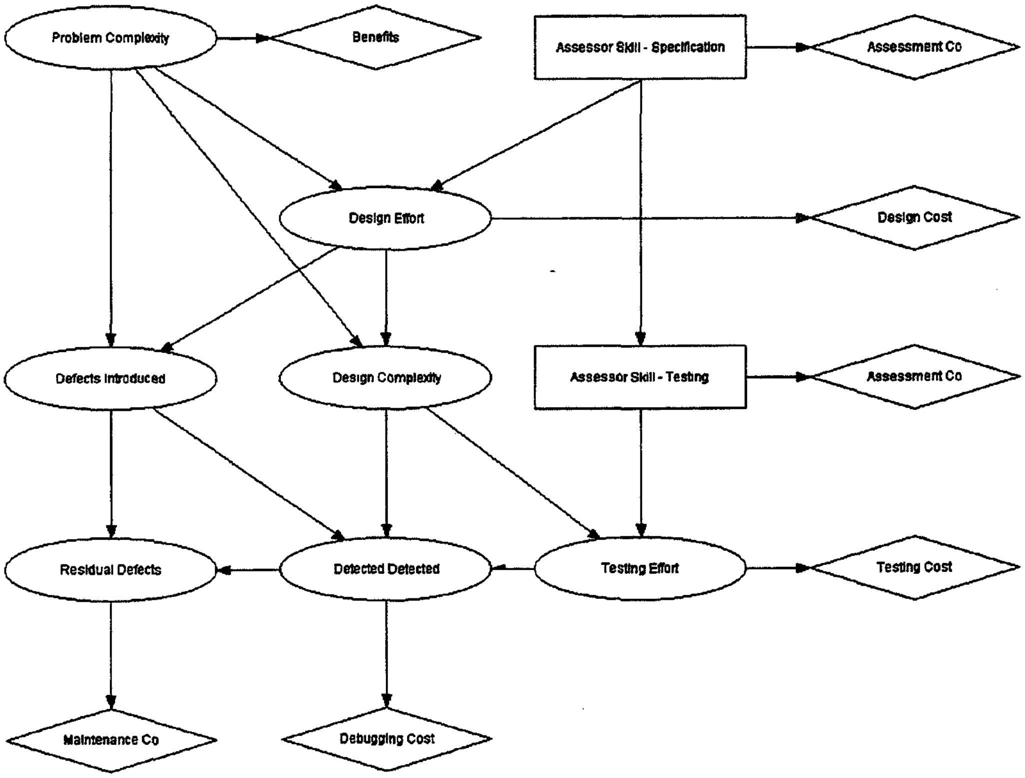


Figure A - BBN Topology

The topology of the Defect Density BBN is shown in Figure A. The ellipses represent 'chance' variables, the rectangles show the 'decisions', the diamonds represent 'utility' (cost/benefit) variables and the arrows show the flow of information or cause-effect links. The variables represented are measured on ordinal, subjective, scales. Subjective scales are used to make the model simpler; there is no theoretical impediment to modelling ratio scales and continuous variables. Each variable has the following states: very-high, high, medium, low, very low or none (optional for some variables). The probabilities attached to each of these states is determined from an analysis of the literature or common-sense assumptions about the direction and strength of relations between variables.

The BBN can be explained in two stages. The first stage covers the life-cycle processes of specification, design or coding and the second stage covers testing. In Figure A p*roblem complexity* represents the degree of complexity inherent in the set of problems to be solved by development. We can think of these problems as being discrete functional requirements in the specification. Solving these problems accrues *benefits to* the user. At the specification stage a project manager assesses the complexity of the problems and assigns *design effort* accordingly. The skill with which this is done is denoted by the variable: *assessor skill—specification*. This assessment process could involve formal measurement, using function points for example, subjective judgement or some combination of both. Assessing the complexity of the problem accrues an *assessment cost—specification*. Any mismatch between the problem complexity and design effort is likely to cause the introduction of defects and a greater design complexity. Hence the arrows between *design effort, problem complexity*, *introduced defects* and *design complexity*. For example an optimistic project manager may allocate a small amount of design effort to a complex problem simply because the complexity was underestimated during assessment of the specification. Applying design effort incurs a *design cost*.

In Figure A the testing stage follows the design stage. Here *design complexity* is assessed by the project manager in order to gauge the amount of *testing effort* to allocate. This decision is represented by the *assessor skill—testing variable*. This is similar to the specification assessment process in that the project manager may measure the design complexity directly using appropriate static or dynamic metrics or will make a guess based on intuition and experience. The extent to which either of these measure precisely the actual design complexity will be uncertain. Doing the assessment will incur *assessment cost—testing*. Ideally any testing effort allocated would match that required by the design complexity. However in practice the testing effort actually allocated may be much less, whether by intent or accident. The mismatch between testing effort and design complexity will influence the number of *defects detected*, which is bounded by the number introduced. Fixing these defects during testing incurs a *de-bugging cost*. The difference between the defects detected and defects introduced is the *residual defects* count. Any residual defects will be released with the product and may increase the *maintenance costs*, incurred by the user and maintainer.

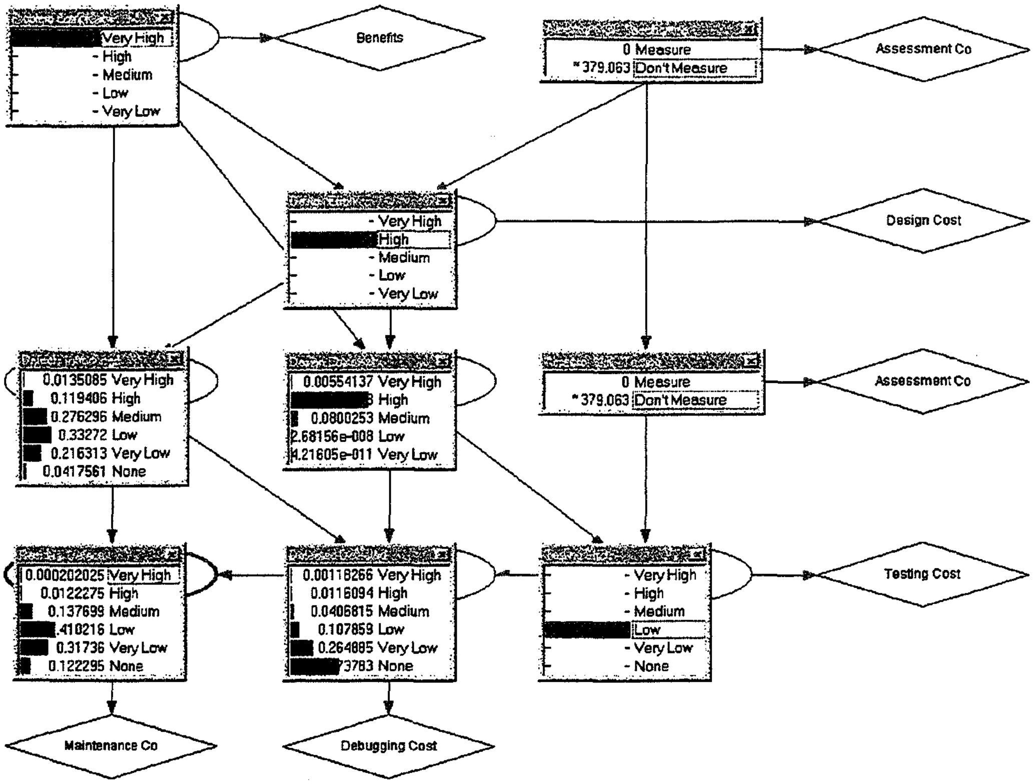


Figure B - Is Bigger Better? Dilemma

Figure B shows the execution of the defect density BBN model for the 'Is Bigger Better?' dilemma using the *Hugin Explorer* tool. Each of the decision and chance variables is shown as a window with a histogram of the predictions made based on the facts entered. The scenario runs as follows. A very complex problem is represented as a fact set at 'very high'. Assume the project manager performs no precise estimation on this so the *assessment skill—specification* variable is set to 'no measurement'. This results in the allocation of 'high' *design effort*, rather than 'very high' commensurate with the problem complexity. The model then propagates these 'facts' and predicts the *design complexity* with a peak at 'high' with probability of approx. 90%. The *introduced defects* follows a modal distribution shape with a peak at 'medium' with probability of around 27%. We may also find that the project manger is again optimistic. He does not measure the *design complexity* and allocates a 'low' level of *testing effort*. This results in low levels of *defects detected*, with approximately 60% probability of finding no defects at all. From the predicted values for detected and introduced defects is propagated to predict the residual defects. Residual defects peaks at 'low' with around 40% probability but with a significant tail towards medium and high numbers of residual defects.

From the model we can see a credible explanation for observing large 'modules' with lower defect densities. Under allocation of design effort for complex problems results in more introduced defects and higher design complexity. Higher design complexity requires more testing effort, which is unavailable, leading to less defects being discovered than are actually there. Dividing the small detected defect counts with large design complexity values will result in small defect densities! The model explains the "is bigger better" phenomena without ad-hoc explanation or identification of 'outliers'.

5. *The Way Forward*

At a general level we can see how the use of BBNs and the defect density model provide a significant new approach to modelling software engineering processes and artefacts. The dynamic nature of this model provides a way of simulating different events and identifying optimum courses of action based on uncertain knowledge. These benefits are reinforced when we examine how the model explains known results, in particular the 'Is Bigger Better?' dilemma. Our new approach shows how we can build complex webs of interconnection between process, product and resource factors in a way hitherto unachievable. We also should how we can integrate uncertainty and subjective criteria into the model without sacrificing rigour and illustrate how decision-making throughout the development process influences the quality achieved.

The benefits of this new approach are:

- it is more useful for project management than outlier analysis and classical statistics
- it incorporates current research ideas and experience
- it can be used to train managers and enable comparison of different decisions by simulation and what-if analyses
- it integrates a form of cost and quality forecasting

So far we have explained historical results rather than real projects. Much work remains to be done to:

- provide guidelines on how to apply the approach to specific situations
- develop a modular approach where whole development processes can be modelled using linked BBNs
- assess the validity of the model by testing its predictions on real projects

We have embarked on the above tasks in the area of safety cases in the CEC ESPRIT project SERENE (Safety and Risk Evaluation using Bayesian Nets) and will be improving it for statistical software process control in the IMPRESS (Improving the Software Process using Bayesian Nets) project funded by UK EPSRC. We will be applying the defect density BBN model to a project with Ericsson Radio Systems in Sweden and are working with the UK Defence Research Agency (DRA) to develop BBNs for procurement processes.

Acknowledgements

This work was supported in part by the ESPRIT projects SERENE and DEVA.

References

[Akiyama 1971] Akiyama, F 'An example of software system debugging', Inf Processing 71, 353-379 1971

[Bache and Bazzana 1993] R.Bache, G.Bazzana (1993) Software metrics for product assessment , McGraw Hill, London.

[Basili and Perricone 1984] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation", Communications of the ACM, 1984, pp.42-52.

[Compton and Withrow 1990] , 'Prediction and control of Ada software defects', Proc 2nd Annual Oregon Workshop on Software Metrics, March 1990.

[Cusumano 1991] Cusumano, MA, Japan's Software Factories, Oxford University Press, 1991.

[Fenton and Pfleeger 1996] Fenton NE, Pfleeger SL Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, 1996.

[Gaffney 1984] J.E.Gaffney, JR., "Estimating the Number of Defects in Code", IEEE Trans. Software Engineering, Vol.SE-10, No.4, 1984

[Hatton 1994]Hatton, L. (1994). C and Safety Related Software Development: Standards, Subsets, testing, Metrics, Legal issues. McGraw-Hill.

[Hugin 1996] Hugin Expert A/S Niels Jernes Vej 10 DK - 9220 Aalborg, Denmark

[Koga 1992] Koga K, 'Software Reliability Design Method in Hitachi', Proceedings of the 3rd European Conference on Software Quality, Madrid, 1992

[Lipow 1982] M.Lipow, "Number of Defects per Line of CODE", IEEE Trans. Software Engineering, Vol.SE-8, No.4, 1982, 437-439, 1982

[Moeller and Paulish 1993] K.H. Moeller, D. Paulish, 'An emprirical investigation of software defect distribution', Proc 1st Intl Software Metrics Symp, IEEE CS Press, 82-90, 1993

[Neil 1992] Neil, M.D. "Multivariate Assessment of Software Products". Journal of Software Testing, Verification and Reliability, Vol 1(4), pp 17-37, 1992.

[Ottenstein 1979] Ottenstein LM, 'Quantitative estimates of debugging requirements', IEEE TSE 5(5), 504-514, 1979

[Pearl 1988] Pearl J, Probabilistic reasoning in intelligent systems, Morgan Kaufmann, Palo Alto, CA, 1988.

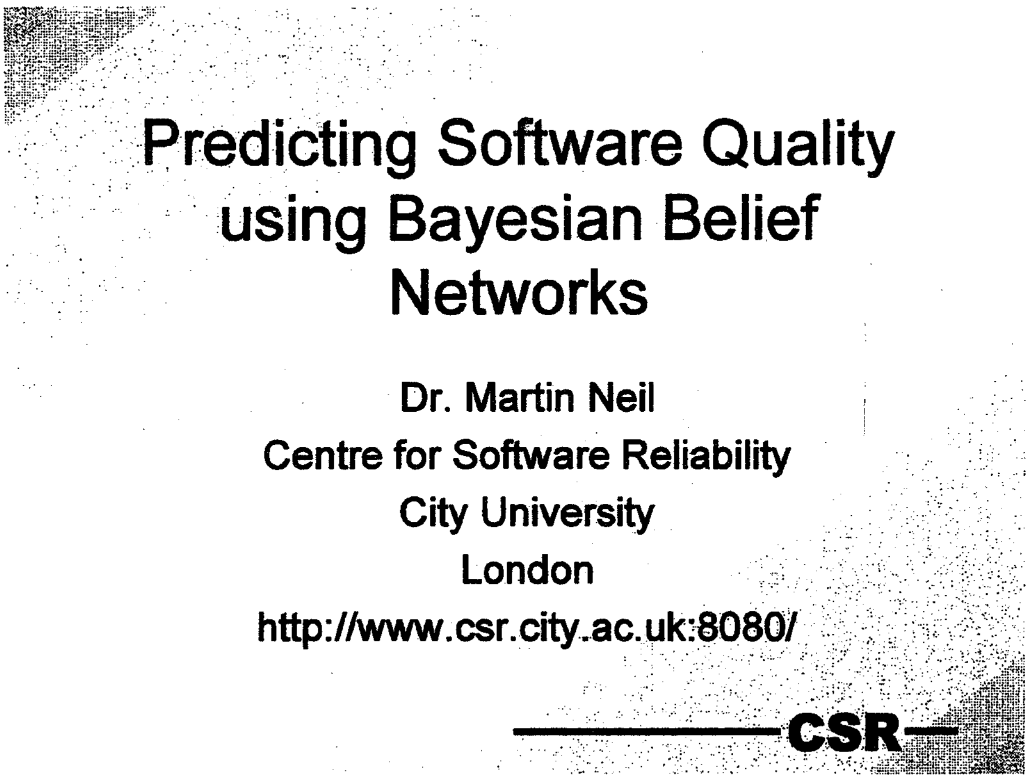
[Shen et al 1983] Shen VY, Conte SD, Dunsmore H, `Software science revisited: a critical analysis of the theory and its empirical support', IEEE Trans Soft Eng, SE-9(2), 1983, 155-165.

[Shen 1985] Shen VY, Yu T, Thebaut SM, Paulsen LR, 'Identifying error-prone software - an empirical study', IEEE Trans Soft Eng SE-11(4) 1985, 317-323.

[Veevers and Marshall 1994] Veevers A and Marshall AC, A relationship betwen software coverage metrics and reliability', J Software Testing, Verification and Reliability, 4, 3-8, 1994

[Yasuda 1989] Yasuda, K.'Software Quality Assurance Activities in Japan. In Japanese Perspectives in Software Engineering,.187-205, Addison-Wesley, 1989.

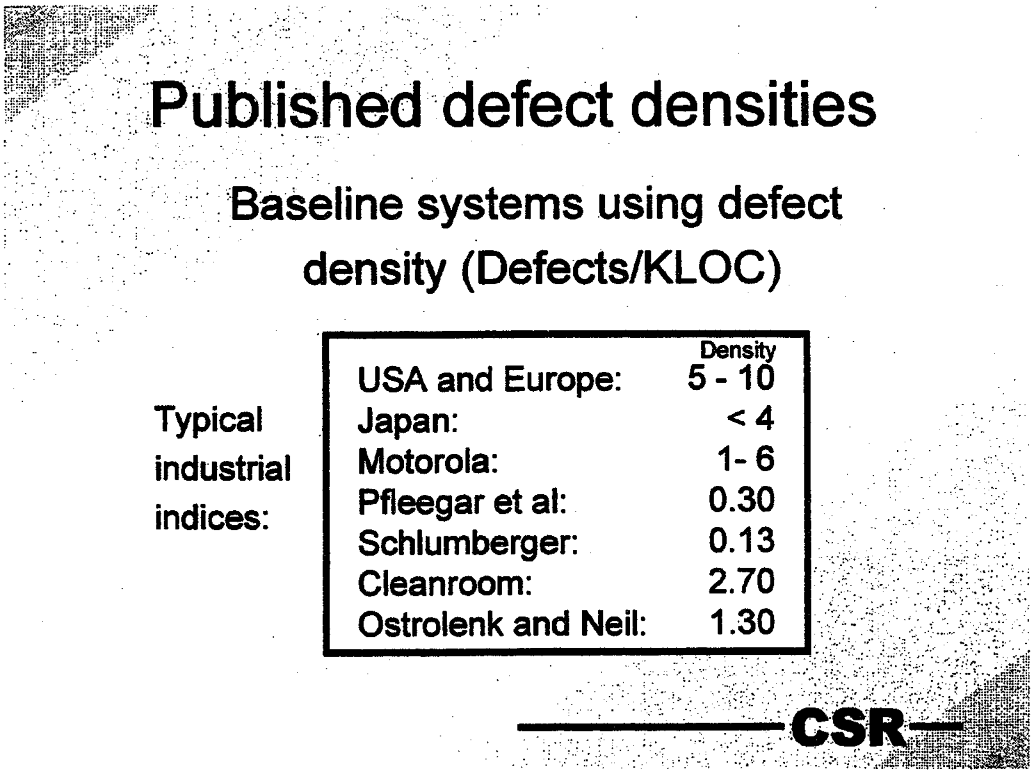
[Friedman and Voas, 1995] M. A. Friedman and J. M. Voas Software Assessment: Reliability, Safety and Testability. John Wiley and Sons, 1995.



Predicting Software Quality using Bayesian Belief Networks

Dr. Martin Neil
Centre for Software Reliability
City University
London
http://www.csr.city.ac.uk:8080/

CSR



Published defect densities

Baseline systems using defect density (Defects/KLOC)

Typical industrial indices:

	Density
USA and Europe:	5 - 10
Japan:	< 4
Motorola:	1- 6
Pfleegar et al:	0.30
Schlumberger:	0.13
Cleanroom:	2.70
Ostrolenk and Neil:	1.30

CSR

Lines of Code Models

- Akiyama

  $D = 4.86 + 0.018\,L$

- Gaffney

  $D = 4.2 + 0.0015\,L^{3/4}$

  (optimum module size 877 LOC)

- Compton and Withrow

  $D = 0.069 + 0.00516\,L + 0.00000047\,L^2$

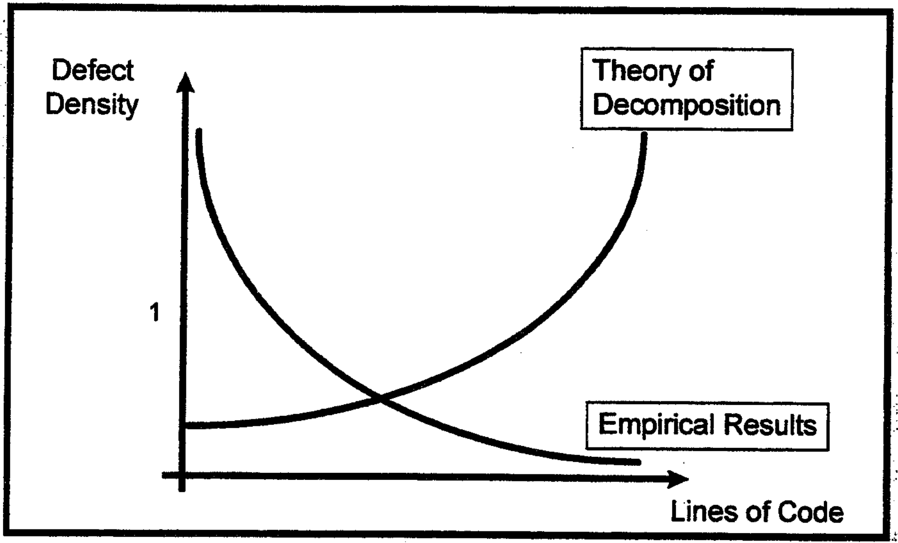
  (optimum module size 83 Ada LOC)

D: Defects

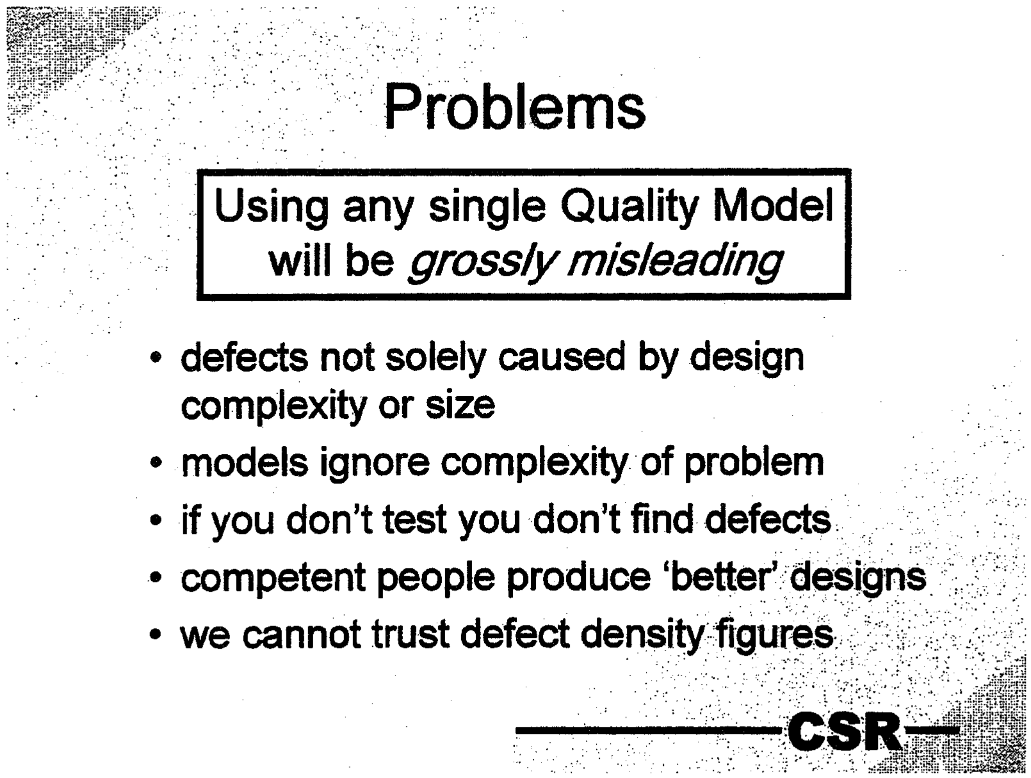
L: Lines of Code

CSR

Is Bigger Better?



CSR

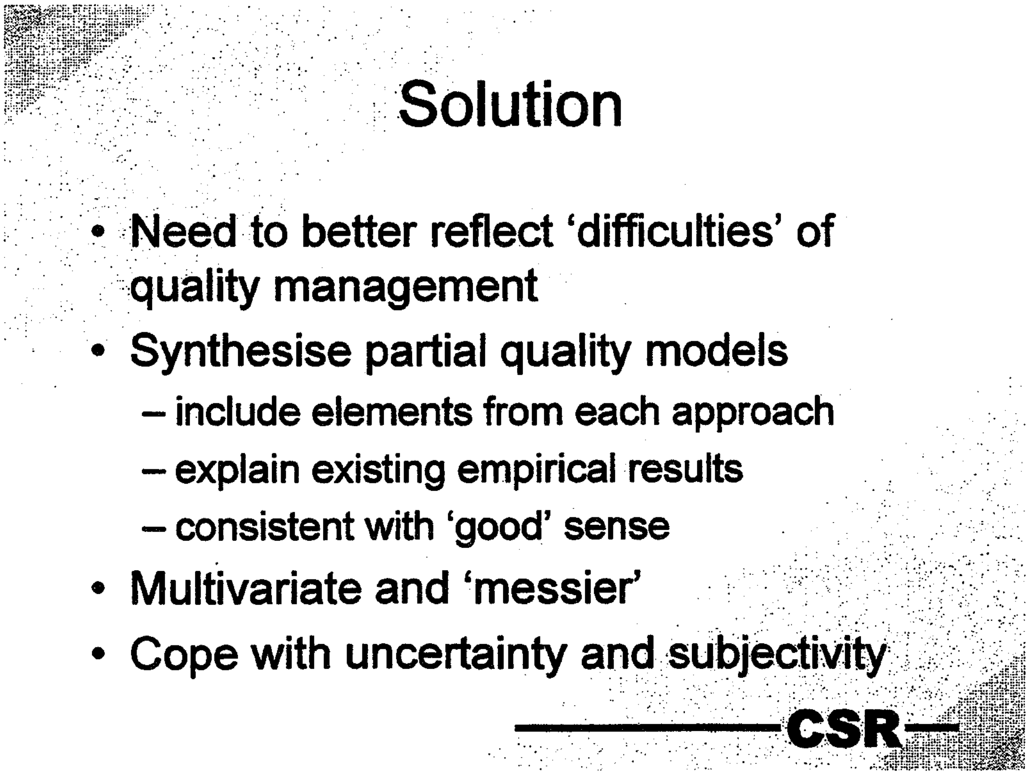


Problems

Using any single Quality Model will be *grossly misleading*

- defects not solely caused by design complexity or size
- models ignore complexity of problem
- if you don't test you don't find defects
- competent people produce 'better' designs
- we cannot trust defect density figures

CSR



Solution

- Need to better reflect 'difficulties' of quality management
- Synthesise partial quality models
  - include elements from each approach
  - explain existing empirical results
  - consistent with 'good' sense
- Multivariate and 'messier'
- Cope with uncertainty and subjectivity

CSR

Bayesian Belief Networks (BBNs)

- Consists of three major components:
  - graphical model
  - conditional probability tables modelling prior probabilities and likelihoods
  - Bayes' theorem applied recursively to propagate data through network
- Graph topology models cause-effect reasoning structures

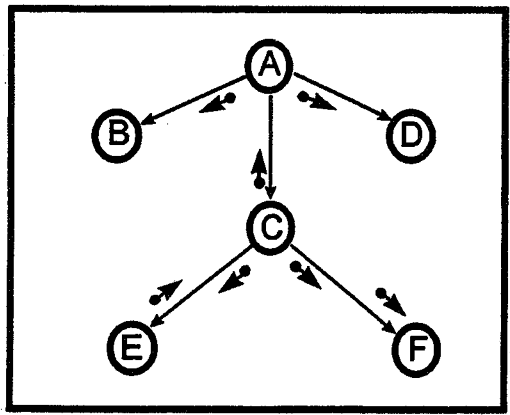
CSR

Propagation

Data entered
E updates parent, C

First activation
C updates neighbours, A, F

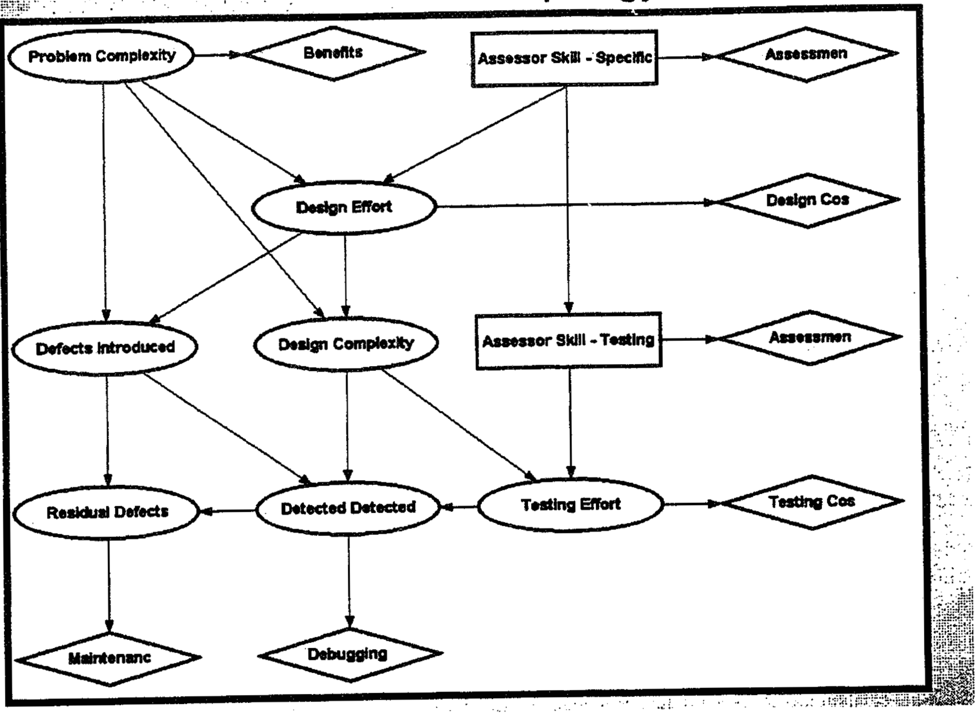
Second activation
root nodes, A, C update children, B, D, E, F



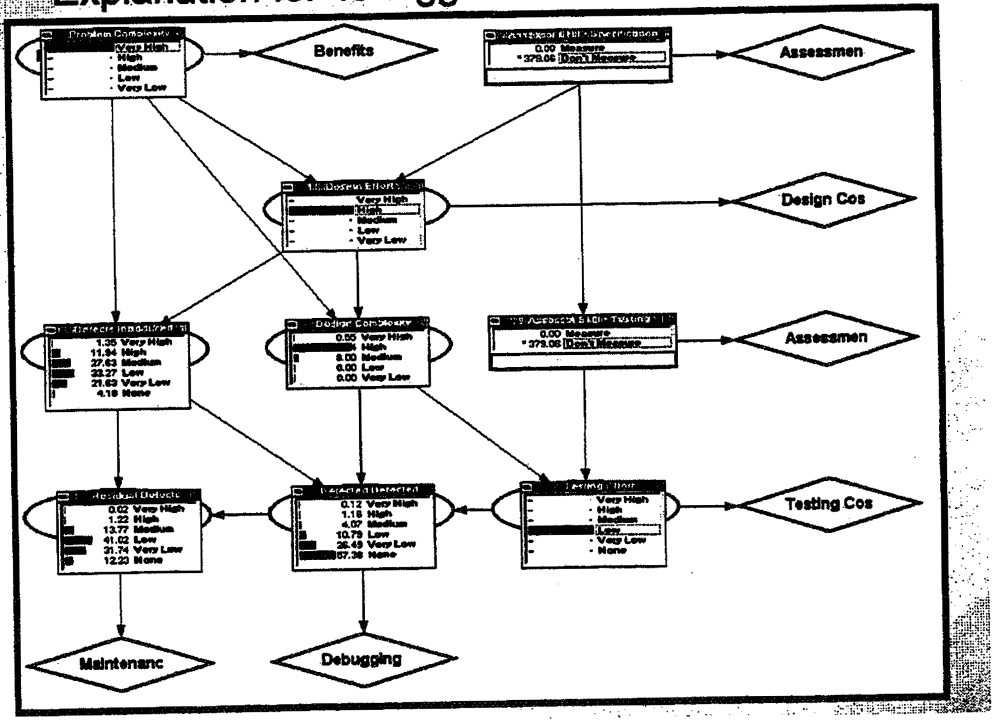
Algorithm based on award winning work by Lauritzen and Spiegelhalter

CSR

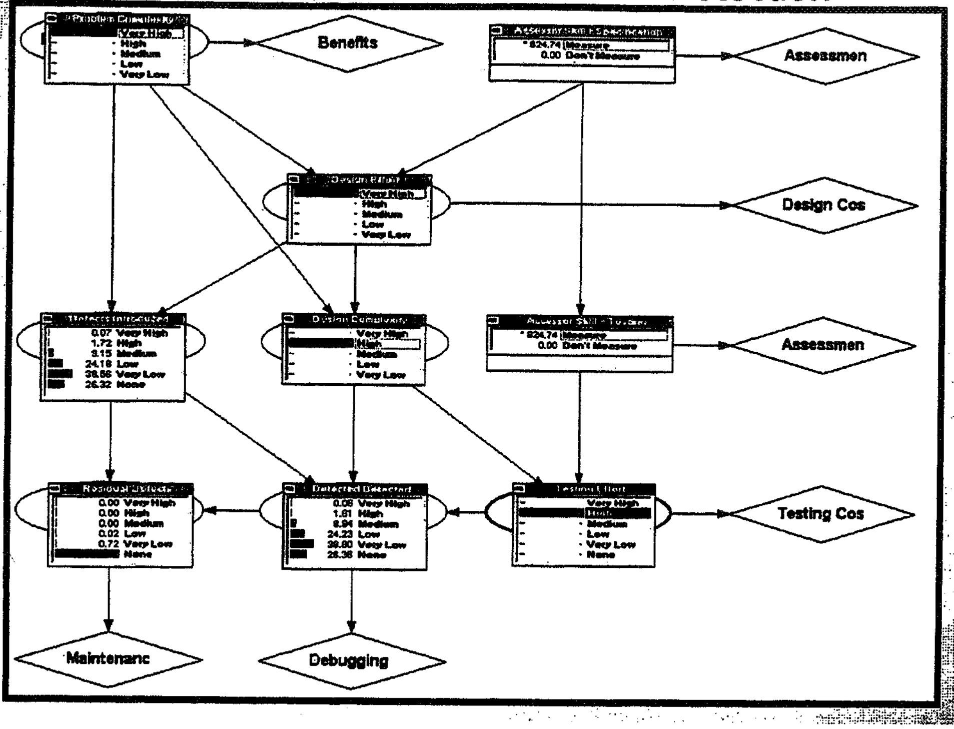
Defects BBN Topology



Explanation for 'Is Bigger Better?' Phenomena



Effective Defect Prevention and Detection



Conclusions

- Equivocal results when partial quality models applied
- New model is synthesis of partial models
- Bayesian Belief Networks offers technology to implement new model
- Coherent model of expertise - empirical validation remains to be done

CSR

DATA COLLECTION DEMONSTRATION and SOFTWARE RELIABILITY MODELING FOR a MULTI-FUNCTION DISTRIBUTED SYSTEM

Dr. Norman F. Schneidewind

Naval Postgraduate School
Code SM/Ss
Monterey, California 93943

Voice: 408-656-2719
Fax: 408-656-3407

Internet: schneidewind@nps.navy.mil

S12-61

415656

360727

p30

NEED FOR A Multi Function Distributed System (MFDS) MODEL

Popular software reliability models treat software as a single entity and model the failure process in accordance with this perspective. However in a MFDS, with multiple clients and servers, this approach is not applicable. Consequently a software reliability model was developed that takes into account the fact that not all software defects and failures result in *system failures* in a client-server system. In this model there are critical clients and servers: clients and servers with critical functions (e.g., network communication) that must be kept operational for the system to survive. There are also non-critical clients and servers with non-critical functions (e.g., email). These clients and servers also act as backups for critical clients and servers, respectively. The system does not fail unless all non-critical clients fail and one or more critical clients fail, or all non-critical servers fail and one or more critical servers fail.

The Marine Corps Tactical System Support Activity (MCTSSA) required the development of such a model because the MFDS is the type of system that is developed by this agency, where valid predictions of software reliability are important for evaluating the reliability of systems that will be deployed in the field.

CLIENT-SERVER SOFTWARE RELIABILITY PREDICTION

This section provides an introduction to client-server software reliability prediction and provides definitions of several important terms. Too often the assumption is made, when doing software reliability modeling and prediction, that the software involves a *single* node. The reality in today's increasing use of *multi* node client-server systems is that there are multiple entities of software that execute on multiple nodes that must be modeled in a *system* context, if realistic reliability predictions and assessments are to be made. For example if there are N_c clients and N_s servers in a client-server system, it is not necessarily the case that a software failure in any of the N_c clients or N_s servers , which causes the node to fail, will cause the *system* to fail. Thus, if such a system were to be modeled as a single entity, the predicted reliability would be much lower than the true reliability because the prediction would not account for *criticality* and *redundancy*. The first factor accounts for the possibility that the survivability of some clients and servers will be more critical to continued *system* operation than others, while the second factor accounts for the possibility of using redundant nodes to allow for system recovery should a critical node fail. To address this problem, we must identify which nodes -- clients and servers -- are critical and which are not critical. We use the following definitions:

Node: A hardware element on a network, generally a computer, that has a network interface card installed [NOV95].

Client: A node that makes requests of servers in a network or that uses resources available through the servers [NOV95].

Server: A node that provides some type of network service [NOV95].

Client-Server Computing: Intelligence, defined either as processing capability or available information, is distributed across multiple nodes. There can be various degrees of allocation of computing function between the client and server, from one extreme of an application running on the client but with requests for data to the server to the other extreme of a server providing centralized processing (e.g., mail server) and sharing information with the clients [NOV95]. The terms *client-server computing* and *distributed system* are used synonymously.

Critical function: An application function that must operate for the duration of the mission, in accordance with its requirement, in order for the system to achieve its mission goal (e.g., the requirement states that a military field unit must be able to send messages to headquarters and receive messages from headquarters during the entire time that a military operation is being planned). This type of function operates in the *network mode*, which means that the application requires more than a single client to perform its function; thus client to server or client to client communication is required.

Non-critical function: An application function that does not have to operate for the duration of the mission in order for the system to achieve its mission goal (e.g., it is not necessary to perform word processing during the entire time that a military operation is being planned). Often this type of function operates in the *standalone mode*, which means that a single client performs the application function; thus client to server or client to client communication is not required, except for the possible initial downloading of a program from a file server or the printing of a job at a print server.

Critical clients and servers: Nodes with critical functions, as defined above. These nodes must be kept operational for the system to survive, either by incurring no failures or by reconfiguring non-critical nodes to operate as critical nodes.

Non-critical clients and servers: Nodes with non-critical functions, as defined above. These nodes also act as backups for the critical nodes, should the critical nodes fail.

Software Defect: *Any* undesirable deviation in the operation of the software from its intended operation, as stated in the software requirements.

Software Failure: A defect in the software that *causes* a node (either a client or a server) in a client-server system to be unable to perform its required function within specified performance requirements (i.e., a node failure).

System Failure: The state of a client-server system, which has experienced one or more node failures, wherein there are insufficient numbers and types of nodes available for the system to perform its required functions within specified performance requirements.

MODEL FORMULATION

By defining *System Nodes*, *Node Failure Probabilities*, and *Failure States*, the user will be able to compute the probability of system failure *given* that a node failure has occurred. Start by defining the number and type of MFDS nodes as follows:

System Nodes

N_{cc}: Number of Critical Client nodes.
$N_{nc}(t)$: Number of Non-Critical Client nodes.
N_{cs}: Number of Critical Server nodes.
$N_{ns}(t)$: Number of Non-Critical Server nodes.
The sum of these nodes should equal the total number of nodes:

$$N=N_{cc}+N_{nc}(t)+N_{cs}+N_{ns}(t). \qquad (1)$$

As long as the system survives, N_{cc} and N_{cs} are constants because a failure of a critical node will result in a non-critical node replacing it, if there is a non-critical node available. A change in software configuration may be necessary on the former non-critical node in order to run the failed critical node's software. If a critical node fails, the system fails, *if there are no non-critical nodes available* on which to run the failed critical node's software.

In contrast, $N_{nc}(t)$ and $N_{ns}(t)$ are decreasing functions of operating time because these nodes replace failed critical nodes, and are not themselves replaced, where $N_{nc}(0)$ is the number of non-critical clients and $N_{ns}(0)$ is the number of non-critical servers at the start of system operation, respectively. In addition, if a non-critical node fails, the function that had been operational on the failed node can be continued on another node of this type and the system can continue to operate in a degraded state. When either a non-critical node replaces a critical node or a non-critical node fails, $N_{nc}(t)$ or $N_{ns}(t)$ is decreased by one, as appropriate.

Node Failure Probabilities

We must also account for the following node failure probabilities:

p_{cc}: probability of a software defect causing a critical client node to fail.
p_{nc}: probability of a software defect causing a non-critical client node to fail.
p_{cs}: probability of a software defect causing a critical server node to fail.
p_{ns}: probability of a software defect causing a non-critical server node to fail.

These probabilities are important to know individually in the analysis; they are also important in the computation of the probability of *system failure*.

The general function for the probability of system failure, *given a node failure*, is the following:

P_{sys}/node fails=$f(N_{cc}, p_{cc}, N_{nc}, p_{nc}, N_{cs}, p_{cs}, N_{ns}, p_{ns})$ (2)

Equation (2) means that the probability of a system failure, *given a node failure*, is dependent on the four node counts and the corresponding four failure probabilities. The four probabilities are computed from data that is derived from a defect database (defect descriptions, defect classifications, and administrative information) as follows:

$p_{cc}=\sum_I f_{cc}(I)/D$, where $f_{cc}(I)$ is the critical client node failure count in interval I; (3)
$p_{nc}=\sum_I f_{nc}(I)/D$, where $f_{nc}(I)$ is the non-critical client node failure count in interval I; (4)
$p_{cs}=\sum_I f_{cs}(I)/D$, where $f_{cs}(I)$ is the critical server node failure count in interval I; (5)
$p_{ns}=\sum_I f_{ns}(I)/D$, where $f_{ns}(I)$ is the non-critical server node failure count in interval I; (6)
and the total defect count across all intervals is $D=\sum_I d(I)$, (7)

where I is the identification of an interval of operating time of the software and d(I) is the total defect count in interval I.

In a specific application, Boolean expressions (i.e. expressions containing *AND*, *OR*, and *NOT*, logic operations) are used to search the defect database and extract the failure counts (e.g., $f_{cc}(I)$) that are used to compute equations (3)-(6). These expressions specify the conditions that qualify a defect as a node failure (e.g., defect that is a General Protection Fault that affects network operations on a Windows-based system).

Failure States

Next we need to know that at a given instant in test or operational time t, a MFDS may be in one of three failure states that pertains to the survivability of the system, as follows, in decreasing order of capability:

Degraded - Type 1: A software defect in a non-critical node causes the node to fail. As a result, the system operates in a degraded state, with one less non-critical node. No reconfiguration is necessary because the failed node is not replaced.

Degraded - Type 2: A software defect in a critical node causes the node to fail. As a result, the system operates in a degraded state, but one that is more severe than *Type 1*, because there would be both a temporary loss of one critical node during reconfiguration and a permanent loss of one non-critical node (i.e., one of the non-critical nodes takes over the function of the failed critical node). Under certain conditions -- see Table 1 -- this type of node failure can cause a system failure.

The current version of the model assumes that node failures are not recoverable on the node where the failure occurred, *during the* mission. The next version of the model will contain a repair function to account for the case where a node failure is repaired and the node is put back into operation during the mission.

System Failure: The system fails under the following conditions: 1) all non-critical clients fail **and** one or more critical clients fail, **or** 2) all non-critical servers fail **and** one or more critical servers fail. The reason for this failure event formulation is that, in the event of a failed critical node, a non-critical node can be substituted, possibly with a different software configuration. However, if all non-critical clients (servers) fail, and one or more critical clients (servers) fail, there would be no non-critical clients (servers) left to take over for the failed critical clients (servers).

The failure states are summarized in Table 1.

System Failure Probability

Having equations (3)-(6) for the node failure probabilities in hand, the model applies them to computing the probability of system failure -- equation (12). The intermediate equations leading up to equation (12) follow:

The probability that **one or more** critical clients N_{cc} fail, given that the software fails, is:

$$P_{cc}=1-(1-p_{cc})^{Ncc} \qquad (8)$$

The probability that **all** non-critical clients $N_{nc}(t)$ have failed by time t, given that the software fails, is:

$$P_{nc}(t)=(p_{nc})^{Nnc(t)} \qquad (9)$$

The probability that **one or more** critical servers N_{cs} fail, given that the software fails, is:

$$P_{cs}=1-(1-p_{cs})^{Ncs} \qquad (10)$$

The probability that **all** non-critical servers $N_{ns}(t)$ have failed by time t, given that the software fails, is:

$$P_{ns}(t)=(p_{ns})^{Nns(t)} \qquad (11)$$

Equations (8) and (9) assume that client failures are independent (i.e., one type of node failure does not cause another type of node failure). This is the case because a failure in one client's software would not cause a failure in another client's software. However it is possible that a failure in server software could cause a failure in client software, such as a client accessing a server that has corrupted data. Also, equations (10) and (11) assume that server failures are independent. This is the case because a failure in one server's software would not cause a failure in another server's software. However it is possible that a failure in client software could cause a failure in server software, such as a client with corrupted data accessing a server. No case of client failures that were caused by server failures nor of the converse have been found in the *LOGAIS* database. Of course, this does not mean that these events could not happen in general. To account for the possibility of these events, we would need to include the conditional probability of a client failure, given a server failure, and the converse. This model formulation is beyond the scope of this handbook and will be included in the next version of the model.

Combining (8), (9), (10), and (11), the probability of a system failure by time t, given that a node fails, is:

$$P_{sys}/\text{node fails}=[P_{cc}][P_{nc}(t)]+[P_{cs}][P_{ns}(t)]=[1-(1-p_{cc})^{Ncc}][(p_{nc})^{Nnc(t)}]+[1-(1-p_{cs})^{Ncs}][(p_{ns})^{Nns(t)}] \qquad (12)$$

and the probability of a node failure due to software is:

$$p_{sw}=p_{cc}+p_{nc}+p_{cs}+p_{ns} \qquad (13)$$

Time to Failure Prediction

In order to make *Time to Failure* predictions for each of the four types of node failures, the user first analyzes the defect data to determine what type of software defects could cause each of the four types of node failures; then the user partitions the defect data accordingly. More will be said about this process in the *Application of Model* section. Next the user applies equation (14) of the *Schneidewind Software Reliability Model* [AIA93, KEL95, LYU 96, SCH92, SCH93] to make each of the four predictions, using the *SMERFS* software reliability tool [FAR93]. In equation (14), $T_f(t)$ is the predicted time (intervals) until the next F_t failures (one or more) occur, α and β are failure rate parameters, s is the first interval where the observed failure data is used, t is the current interval, and $X_{s,t}$ is the cumulative number of failures observed in the range s,t.

$$\mathbf{T_F(t)=[(\log[\alpha/(\alpha-\beta(X_{s,t}+F_t))])/\beta]-(t-s+1)}$$
$$\mathbf{for\ (\alpha/\beta)>(X_{s,t}+F_t)} \qquad (14)$$

Time to Failure predictions are made for critical clients, non-critical clients, critical servers, and non-critical servers. As the predicted failure times are recorded, the user observes whether the condition for system failure, as defined previously, has been met. If this is the case, a predicted system failure is recorded. Thus, in addition to monitoring the types of predicted failures (e.g., critical client), the process also involves monitoring $N_{nc}(t)$ and $N_{ns}(t)$ to identify the time t when either is reduced to zero, signifying that the supply of non-critical clients or non-critical servers has been exhausted. In this situation, a failure of a critical client or critical server, respectively, will result in a system failure. Thus the user predicts a system failure when the following expression is true (where "$\wedge$" means "AND" and "$\vee$" means "OR"):

$$((\text{Predict critical client failure})\wedge(N_{nc}(t)=0))\vee((\text{Predict critical server failure})\wedge(N_{ns}(t)=0)) \qquad (15).$$

If the predictions produce multiple node failures in the same interval (e.g., critical client and critical server), the user records multiple failures for that interval.

APPLICATION OF THE MODEL

Analysis of the Defect and Failure Data

In this example the user applies the software reliability model to the Marine Corps LOGAIS system -- a client-server logistical support system. In this system it is important that the reliability specification distinguish between failure states *Degraded-Type 1*, *Degraded-Type 2*, and *System Failure*, as previously defined (i.e., distinguish between node failures that cause performance degradation but allow the system to survive, and node failures that cause a system failure). This distinction is made when analyzing the system's defect data. The defect data used in the example are from the LOGAIS defect database, using the *Defect Control System* (DCS), a defect database management system which was used on the *LOGAIS* project [MHB96, MTP96].

In this Windows-based client-server system, the types of clients and servers that were previously defined are used, with corresponding types of defects and failures, as identified in the defect database [MHB96, MTP96]. The following short-hand notation for identifying the attributes of the defect database is used:

o S: Software Defect
o G: General Protection Fault (GPF)
o N: Network Related Failure
o C: System Crash

The LOGAIS defect database is queried in order to identify the software defects that qualify as node failures. The following Boolean expressions, corresponding to the four types of node failures, are used:

1. Critical Client Failure: COUNT as failures WHERE (S$\wedge$G$\wedge$N$\wedge$*not*C). A *GPF* causes a node failure (*Degraded-Type 2*) on a critical client, a client which must maintain communication with other nodes on the network (*Network Mode*), and the failure does not cause a *System Crash* (loss of server).

2. Non-Critical Client Failure: COUNT as failures WHERE (S$\wedge$G$\wedge$*not*N$\wedge$*not*C). A *GPF* causes a node failure (*Degraded-Type 1*) on a non-critical client, a client which does not have to maintain communication with other nodes on the network (*Standalone Mode*), and the failure does not cause a *System Crash* (loss of server).

3. Critical Server Failure: COUNT as failures WHERE (S$\wedge$*not*G$\wedge$N$\wedge$C). A *System Crash* causes a node failure (*Degraded-Type 2*) on a critical server, a server which must maintain communication with other nodes on the network (*Network Mode*), and the failure is not a *GPF;* it is more serious, resulting in the loss of a server.

4. Non-Critical Server Failure: COUNT as failures WHERE (S$\wedge$*not*G$\wedge$*not*N$\wedge$C). A *System Crash* causes a node failure (*Degraded-Type 1*) on a non-critical server, a server which does not have to maintain communication with other nodes on the network, and the failure is not a *GPF;* it is more serious, resulting in the loss of a server.

The above classification associates *GPF* with clients and *System Crash* with servers; it also associates *Network Related Failures* with critical node failures. Note that this is only an example. For other systems, different defect and failure classifications may be appropriate.

The total failure count is obtained by taking the union of expressions 1-4 as follows:

5. Total Failure Count: COUNT as failures WHERE (S$\wedge$((G$\wedge$*not*C)$\vee$(*not*G$\wedge$C))). This expression is used to verify the correctness

of 1-4 because it should equal their sum.

Observed Range and Prediction Range

The major objective of reliability modeling is to predict future reliability over the prediction range of test or operational time of a system. However to do so, there must be a historical record of defects and failures for computing the model parameters and for making the best fit with the historical data; the data is collected during the observed range of test or operational time of a system. The length of the observed range is determined by the amount of data that has been collected prior to making a prediction, while the length of the prediction range is determined by duration of the system's mission. The observed range in this example is 1,50 intervals and the prediction range is 51-61 intervals. These ranges are arbitrary and selected only to illustrate the process. We note that once a system has been tested or operated over the prediction range, there will be observed defects and failures in this range. The observed defects and failures in the prediction range are listed in Table 2. The failure counts corresponding to types 1-5, above, are summarized in Table 3, which shows the empirical probabilities of node failure that are computed using equations (3)--(7) and (13). For example, for critical clients, the computation is 24/4048=.005929. The user should verify the computations for the remaining types of nodes.

Application Predictions

Time to Failure

Using equation (14) and failure data in the observed range 1-50 (not shown) , we made predictions for *Time to Failure*, for t>50 days, for critical clients, non-critical clients, and non-critical servers, in Tables 4, 5 and 6, respectively. The predictions are made for a given numbers of failures (time to one failure for t>50 days, time to two failures for t>50 days, etc.). The predictions are compared with the actual failure data, with the relative error and average relative error for cumulative values shown. In the case of critical servers, there are only two actual failures, both of which occur in the observed range. Only one prediction of *Time to Failure* for **one more failure** could be made at t=50 for critical servers because the predicted remaining failures at t=50 is 1.40 ; therefore, critical server failures are not tabulated. In the case of non-critical nodes, the failure data is sufficiently dense to allow a failure count interval of one day. In the case of critical clients, the failure data was sparse; thus a five day interval was used for prediction, with these predictions converted to the one day intervals shown in Table 4. We note that predictions are difficult to make with this type of data because the defects and failures are not recorded in CPU execution time. Rather they are recorded in calendar time in batches, as shown in the Table 2, based on administrative convenience. Many of these batches are submitted at the end of a workday. This time becomes the "submit date".

Using the data in Tables 4-6, we merge and sequence the various types of failure predictions in Table 7. The purpose of this table is to construct the scenario of failures and surviving non-critical nodes so that the time of *System Failure* can be predicted. The table shows that seven node failures (i.e., the sequence NS, NC, NC, CC, NC, NC, CC) are predicted to occur before the system is predicted to fail. This occurs at t=61.07 days when there are no non-critical clients available and a critical client fails. No critical server failures are shown in this table because the prediction of *Time to Failure* of 99.35 days cumulative is beyond the prediction range of interest in this example.

Using the data in Tables 4-6, we merge and sequence the various types of actual failures in Table 8. Similar to Table 7, the purpose of this table is to construct the scenario of actual failures and surviving non-critical nodes so that the actual time of *System Failure* can be determined and compared with the predicted values. As in the case of the predictions, this table shows that seven node failures (i.e., the sequence NC, NS, NC, NC, NC, NC, CC) occur before the system fails. This occurs at t=61days when there are no non-critical clients available and a critical client fails. No critical server failures are shown in this table because they occurred prior to the range of this example.

Probability of System Failure

Lastly, using equation (12), we predict the probability of system failure, given a node failure, in column 5 of Table 9, as the system progresses through the predicted failure scenario that was shown in Table 7. Except for row 2 in Table 9, the actual probability is the same as the predicted probability because the actual failure scenario that was shown in Table 8 produces the same numbers of non-critical clients and servers that are shown in columns 6 and 7, respectively. Because the predicted and actual failure scenarios are identical, except for row 2, the predicted time to failure and type of node failure, columns 1 and 2, respectively, can be compared in with the corresponding actual values in columns 3 and 4, for given probabilities of system failure. These values were reproduced from Tables 7 and 8, respectively. Because for a given P_{sys}*/node fails*, the cumulative time to failure occurs later for the predicted values, the model is a bit optimistic with respect to reality for this example. Note that the in the last row of Table 9 the system has not yet failed. This occurs when a critical client fails at Day 61.07 *predicted* (see Table 7) and at Day 61 *actual* (see Table 8). At this time there are no non-critical clients left to replace the failed critical client.

The significant results that emerge from this analysis are that: 1) The P_{sys}*/node fails* is only significant (.029790) when the supply of both non-critical clients and non-critical servers has been exhausted and 2) P_{sys}*/node fails* is significantly lower than the probability of *any* type of node failure caused by a software defect: p_{sw}=.065705, obtained from equation (13) and computed in Table 3. Thus evaluations of system reliability should recognize that *software failures are not necessarily equivalent to system failures* and that assessments of software reliability that treat every failure as equivalent to a system failure will grossly understate system reliability.

CONCLUSIONS AND FUTURE RESEARCH

Based on the above approach, it appears feasible to develop a system software reliability model for a client-server system. In order to implement the approach, it is necessary to partition the defects and failures into classes that are then associated with critical and non-critical clients and servers. Once this is done, predictions are made of *Time to Failure* for each type; the predictions are classified according to those that would result in a node failure caused by a software defect and those that would result in a system failure caused by a series of software defects. Then the probability of system failure is computed. A significant result of the research is that software failures should not be treated as the equivalent of system failures because to do so would grossly understate system reliability.

In future research we will deal with the problem of how to apply the model to a system that has a large number of nodes. The technique that we described for monitoring the times when predicted node and system failures occur would be cumbersome for a large system. It appears that a program must be written to automate this process. Other possible future research activities include the following: extend the model to include hardware failures; develop measures of performance degradation, as nodes fail; include a node repair rate to reflect the possibility of recovering failed nodes during the operation of the system; apply smoothing techniques, such as the moving average, to mitigate anomalies in calendar time defect data.

References

[AIA93] Recommended Practice for Software Reliability, R-013-1992, American National Standards Institute/American Institute of Aeronautics and Astronautics, 370 L'Enfant Promenade, SW, Washington, DC 20024, 1993.

[FAR93] William H. Farr and Oliver D. Smith, Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide, NAVSWC TR-84-373, Revision 3, Naval Surface Weapons Center, Revised September 1993.

[KEL95] Ted Keller, Norman F. Schneidewind, and Patti A. Thornton "Predictions for Increasing Confidence in the Reliability of the Space Shuttle Flight Software", Proceedings of the AIAA Computing in Aerospace 10, San Antonio, TX, March 28, 1995, pp. 1-8.

[LYU96] Michael R. Lyu (Editor-in-Chief), Handbook of Software Reliability Engineering, Computer Society Press, Los Alamitos, CA and McGraw-Hill, New York, NY, 1995.

[MHB96] MCTSSA Software Reliability Handbook, Norman F. Schneidewind and Judie A. Heineman, Naval Postgraduate School, January 10, 1996.

[MTP96] MCTSSA Software Reliability Engineering Training Plan, Norman F. Schneidewind and Judie A. Heineman, Naval Postgraduate School, January 10, 1996.

[NOV95] Werner Feibel, Novell's Complete Encyclopedia of Networking, Novell Press, San Jose, CA, 1995.

[SCH93] Norman F. Schneidewind, "Software Reliability Model with Optimal Selection of Failure Data", IEEE Transactions on Software Engineering, Vol. 19, No. 11, November 1993, pp. 1095-1104.

[SCH92] Norman F. Schneidewind and T. W. Keller, "Application of Reliability Models to the Space Shuttle", IEEE Software, Vol. 9, No. 4, July 1992 pp. 28-33.

Table 1
Failure States

	Degraded - Type 1	Degraded - Type 2	System Failure
Non-Critical Client	Node Failure	Does Not Apply	Does Not Apply
Critical Client	Does Not Apply	Node Failure(s) and $N_{ac}(t)>0$	Node Failure(s) and $N_{ac}(t)=0$
Non-Critical Server	Node Failure	Does Not Apply	Does Not Apply
Critical Server	Does Not Apply	Node Failure(s) and $N_{as}(t)>0$	Node Failure(s) and $N_{as}(t)=0$

Table 2
Chronological Node Failure Count Database (Sample)

CC: Critical Client Node Failure
NC: Non-Critical Client Node Failure
CS: Critical Server Node Failure
NS: Non-Critical Server Node Failure

Interval	Defect ID	Number	Submit Date	CC	NC	CS	NS
51	2633,2634	2	1/24/95		X		
51	2635,2636,2637,2638	4	1/24/95				X
52							
53	2661,2662,2663,2664	4	1/26/95		X		
54	2641,2644,2645,2669, 2671,2672,2673,3003	8	1/27/95		X		
54	2640,2643,2670,2674, 2675,2676,2783	7	1/27/95				X
55	2450	1	1/30/95		X		
56							
57							
58	2487	1	2/2/95				X
59	2511,2512,2513	3	2/3/95		X		
60							
61	3025,3026,3027,3029	4	2/7/95	X			

Table 3
Summary of Node Failures (4048 Software Defects)

	Number of Failures	Probability
1. Critical Client	24	p_{cc}=.005929
2.Non-Critical Client	83	p_{nc}=.020250
3.Critical Server	2	p_{cs}=.000494
4. Non-Critical Server	158	p_{ns}=.039032
5. Total	267	p_{sw}=.065705

Table 4
Critical Client Predictions Made at Time=50 Days
Observed Range=1,50 Days; Failure Count=11; Prediction Range>50 Days

Predicted			Actual		
Given Number of Failures	Time to Failure (Days)	Cumulative Time to Failure (Days)	Time to Failure (Days)	Cumulative Time to Failure (Days)	Relative Error (Percent)
1	5.19	55.19	11	61	-9.52
2	11.07	61.07	11	61	+.11
3	17.88	67.88	11	61	+11.28
4	25.95	75.95	11	61	+24.51
5	35.86	85.86	36	86	-.16

Average=9.12%

Table 5
Non-Critical Client Predictions Made at Time=50 Days
Observed Range=1,50 Days; Failure Count=36; Prediction Range>50 Days

Predicted			Actual		
Given Number of Failures	Time to Failure (Days)	Cumulative Time to Failure (Days)	Time to Failure (Days)	Cumulative Time to Failure (Days)	Relative Error (Percent)
1	2.41	52.41	1	51	+2.76
2	4.87	54.87	1	51	+7.59
3	7.37	57.37	3	53	+8.25
4	9.92	59.92	3	53	+13.06
5	12.52	62.52	3	53	+17.96

Average=9.92%

Table 6
Non-Critical Server Predictions Made at Time=50 Days
Observed Range=1,50 Days; Failure Count=108; Prediction Range>50 Days

Predicted			Actual		
Given Number of Failures	Time to Failure (Days)	Cumulative Time to Failure (Days)	Time to Failure (Days)	Cumulative Time to Failure (Days)	Relative Error (Percent)
1	1.96	51.96	1	51	+1.88
2	3.93	53.93	1	51	+5.75
3	5.90	55.90	1	51	+9.61
4	7.87	57.87	1	51	+13.47
5	9.84	59.84	4	54	+10.81

Average=8.30%

Table 7
Predicted Time to Failure When Failures are Merged and Sequenced. Observed Range=1,50 Days; Prediction Range=51,61 Days
CC: Critical Client NC: Non-Critical Client NS: Non-Critical Server

Cumulative Time to Failure (Days)	Time to Failure (Days)	Type of Failure	Number of Non-Critical Clients Available	Number of Non-Critical Servers Available
50	1.96		5	1
51.96	.45	NS	5	0
52.41	2.46	NC	4	0
54.87	.32	NC	3	0
55.19	2.18	CC	2	0
57.37	2.55	NC	1	0
59.92	1.15	NC	0	0
61.07		CC	System Failure	

Table 8
Actual Time to Failure When Failures are Merged and Sequenced. Range=51,61 Days
CC: Critical Client NC: Non-Critical Client NS: Non-Critical Server

Cumulative Time to Failure (Days)	Time to Failure (Days)	Type of Failure	Number of Non-Critical Clients Available	Number of Non-Critical Servers Available
50	1		5	1
51	0	NC,NS	4	0
51	2	NC	3	0
53	0	NC	2	0
53	0	NC	1	0
53	8	NC	0	0
61		CC	System Failure	

Table 9
Probability of System Failure

Predicted Cumulative Time to Failure (Days)	Predicted Type of Node Failure	Actual Cumulative Time to Failure (Days)	Actual Type of Node Failure	Probability of System Failure Given a Node Failure	Number of Non-Critical Clients Available	Number of Non-Critical Servers Available
50		50		0.000019	5	1
51.96	NS			0.000494*	5*	0*
52.41	NC	51	NC,NS	0.000494	4	0
54.87	NC	51	NC	0.000494	3	0
55.19	CC	53	NC	0.000506	2	0
57.37	NC	53	NC	0.001087	1	0
59.92	NC	53	NC	0.029790	0	0

*** Applies only to predicted values.**

DATA COLLECTION DEMONSTRATION and SOFTWARE RELIABILITY MODELING FOR a MULTI-FUNCTION DISTRIBUTED SYSTEM

Dr. Norman F. Schneidewind

Code SM/Ss
Naval Postgraduate School
Monterey, CA 93943
Phone: (408) 656-2719
Fax: (408) 656-3407
Email: schneidewind@nps.navy.mil

OUTLINE

o **Client-Server Software Reliability Prediction**
- Criticality and Redundancy

o **Definitions**

o **Model Formulation**
- System Nodes
- Node Failure Probabilities
- Estimating Node Failure Probabilities
- Failure States
- System Failure Probability
- Model Concepts
- Time to Failure Prediction

o **Application of the Model**
- Node Failure Counts
- Defect Database
- Application Predictions
- Application Results
- Probability of System Failure

o **Conclusions**

CLIENT-SERVER SOFTWARE RELIABILITY PREDICTION

Too often the assumption is made, when doing software reliability modeling and prediction, that the software involves either a *single* module or node. The reality in today's increasing use of *multi* node client-server and distributed systems is that there are multiple entities of software that execute on multiple nodes that must be modeled in a *system* context, if realistic reliability predictions and assessments are to be made.

Criticality and Redundancy

o If there are N_c clients and N_s servers in a client-server system, it is not necessarily the case that a software failure in any of the N_c clients or N_s servers will cause the *system* to fail. If such a system were to be modeled as a single entity, the predicted reliability would be much lower than the true reliability because the prediction would not account for *criticality* and *redundancy*.

o The first factor accounts for the possibility that the survivability of some clients and servers will be more critical to continued *system* operation than others.

o The second factor accounts for the possibility of using redundant nodes to allow for system recovery should a critical node fail.

o Identify which nodes -- clients and servers -- are critical and which are not critical.

DEFINITIONS

Critical function: An application function that must operate for the duration of the mission, in accordance with its requirement, in order for the system to achieve its mission goal (e.g., the requirement states that a military field unit must be able to send messages to headquarters and receive messages from headquarters during the entire time that a military operation is being planned).

Usually this type of function operates in the *network mode,* which means that the application requires more than a single client to perform its function; thus client to server or client to client communication is required.

Non-critical function: An application function that does not have to operate for the duration of the mission in order for the system to achieve its mission goal (e.g., it is not necessary to perform word processing during the entire time that a military operation is being planned).

Often this type of function operates in the *standalone mode,* which means that a single client performs the application function; thus client to server or client to client communication is not required, except for the possible initial downloading of a program from a file server or the printing of a job at a print server.

DEFINITIONS (Continued)

Critical clients and servers: Nodes with critical functions, as defined above. These nodes must be kept operational for the system to survive, either by incurring no failures or by reconfiguring non-critical nodes to operate as critical nodes.

Non-critical clients and servers: Nodes with non-critical functions, as defined above. These nodes also act as backups for the critical nodes, should the critical nodes fail.

Software Defect: *Any* undesirable deviation in the operation of the software from its intended operation, as stated in the software requirements.

Software Failure: A defect in the software that causes a node (either a client or a server) in a client-server system to be unable to perform its required function within specified performance requirements (i.e., a node failure).

System Failure: The state of a client-server system, which has experienced one or more node failures, wherein there are insufficient numbers and types of nodes available for the system to perform its required functions within specified performance requirements.

MODEL FORMULATION

System Nodes

N_{cc}: Number of Critical Client nodes.

$N_{nc}(t)$: Number of Non-Critical Client nodes.

N_{cs}: Number of Critical Server nodes.

$N_{ns}(t)$: Number of Non-Critical Server nodes.

where the total number of nodes $N=N_{cc}+N_{nc}(t)+N_{cs}+N_{ns}(t)$.

As long as the system survives, N_{cc} and N_{cs} are constants because a failure of a critical node will result in a non-critical node replacing it, if there is a non-critical node available. A change in software configuration may be necessary on the former non-critical node in order to run the failed critical node's software.

If a critical node fails, the system fails, *if there are no non-critical nodes available* on which to run the failed critical node's software.

In contrast, $N_{nc}(t)$ and $N_{ns}(t)$ are decreasing functions of operating time because these nodes replace failed critical nodes, and are not themselves replaced, where $N_{nc}(0)$ is the number of non-critical clients and $N_{ns}(0)$ is the number of non-critical servers at the start of system operation, respectively.

In addition, if a non-critical node fails, the function that had been operational on the failed node can be continued on another node of this type and the system can continue to operate in a degraded state.

When either a non-critical node replaces a critical node or a non-critical node fails, $N_{nc}(t)$ or $N_{ns}(t)$ is decreased by one, as appropriate.

Node Failure Probabilities

p_{cc}: probability of a software defect causing a critical client node to fail.

p_{nc}: probability of a software defect causing a non-critical client node to fail.

p_{cs}: probability of a software defect causing a critical server node to fail.

p_{ns}: probability of a software defect causing a non-critical server node to fail.

Thus *given a node failure*, we have the following function for the probability of system failure:

P_{sys}/node fails=$f(N_{cc}, p_{cc}, N_{nc}, p_{nc}, N_{cs}, p_{cs}, N_{ns}, p_{ns})$

Estimating Node Failure Probabilities

The four probabilities are estimated from data in a defect database as follows:

$p_{cc}=\sum_i f_{cc}(i)/D$, where $f_{cc}(i)$ is the critical client node failure count in interval i;

$p_{nc}=\sum_i f_{nc}(i)/D$, where $f_{nc}(i)$ is the non-critical client node failure count in interval i;

$p_{cs}=\sum_i f_{cs}(i)/D$, where $f_{cs}(i)$ is the critical server node failure count in interval i;

$p_{ns}=\sum_i f_{ns}(i)/D$, where $f_{ns}(i)$ is the non-critical server node failure count in interval i;

and the total defect count across all intervals is $D=\sum_i d(i)$, where i is the identification of an interval of operating time of the software and d(i) is the total defect count in interval i.

In a specific application, Boolean expressions are used to search the defect database and extract the failure counts (e.g., $f_{cc}(i)$) that are used to compute the above equations. These expressions specify the conditions that qualify a defect as a node failure (e.g., defect that is a General Protection Fault that affects network operations on a Windows-based system).

Failure States

At a given time t, the system can be in one of three failure states that pertains to the survivability of the system, as follows, in decreasing order of capability:

Degraded - Type 1: A software defect in a non-critical node causes the node to fail. As a result, the system operates in a degraded state, with one less non-critical node. No reconfiguration is necessary because the failed node is not replaced.

Degraded - Type 2: A software defect in a critical node causes the node to fail. As a result, the system operates in a degraded state, but one that is more severe than *Type 1*, because there would be both a temporary loss of one critical node during reconfiguration and a permanent loss of one non-critical node (i.e., one of the non-critical nodes takes over the function of the failed critical node). Under certain conditions -- see below -- this type of node failure can cause a system failure.

We assume that node failures are non-recoverable on the node where the failure occurred.

System Failure: The system fails under the following conditions: 1) all non-critical clients fail **and** one or more critical clients fail, **or** 2) all non-critical servers fail **and** one or more critical servers fail. The reason for this failure event formulation is that, in the event of a failed critical node, a non-critical node can be substituted, possibly with a different software configuration. However, if all non-critical clients (servers) fail, and one or more critical clients (servers) fail, there would be no non-critical clients (servers) left to take over for the failed critical clients (servers).

The failure states are summarized in Table 1.

Table 1

Failure States

	Degraded - Type 1	Degraded - Type 2	System Failure
Non-Critical Client	Node Failure	Does Not Apply	Does Not Apply
Critical Client	Does Not Apply	Node Failure and $N_{nc}(t)>0$	Node Failure and $N_{nc}(t)=0$
Non-Critical Server	Node Failure	Does Not Apply	Does Not Apply
Critical Server	Does Not Apply	Node Failure and $N_{ns}(t)>0$	Node Failure and $N_{ns}(t)=0$

System Failure Probability

The probability that **one or more** critical clients N_{cc} fail, given that the software fails, is:

$P_{cc}=1-(1-p_{cc})^{Ncc}$

The probability that **all** non-critical clients $N_{nc}(t)$ have failed by time t, given that the software fails, is:

$P_{nc}(t)=(p_{nc})^{Nnc(t)}$

The probability that **one or more** critical servers N_{cs} fail, given that the software fails, is:

$_{PCs}=1-(1-_{PCs})^{Ncs}$

The probability that **all** non-critical servers $N_{ns}(t)$ have failed by time t, given that the software fails, is:

$P_{ns}(t)=(p_{ns})^{Nns(t)}$

Combining the above four equations, the probability of a system failure by time t, given that a node fails, is:

P_{sys}/node fails$=[P_{cc}][P_{nc}(t)]+[_{PCs}][P_{ns}(t)]=$

$$\underbrace{[1-(1-p_{cc})^{Ncc}][(p_{nc})^{Nnc(t)}]}_{\text{Probability of Client Failure}}+\underbrace{[1-(1-_{PCs})^{Ncs}][(p_{ns})^{Nns(t)}]}_{\text{Probability of Server Failure}}$$

Probability of a Node Failure Due to Software

$p_{sw}=p_{cc}+p_{nc}+p_{cs}+p_{ns}$

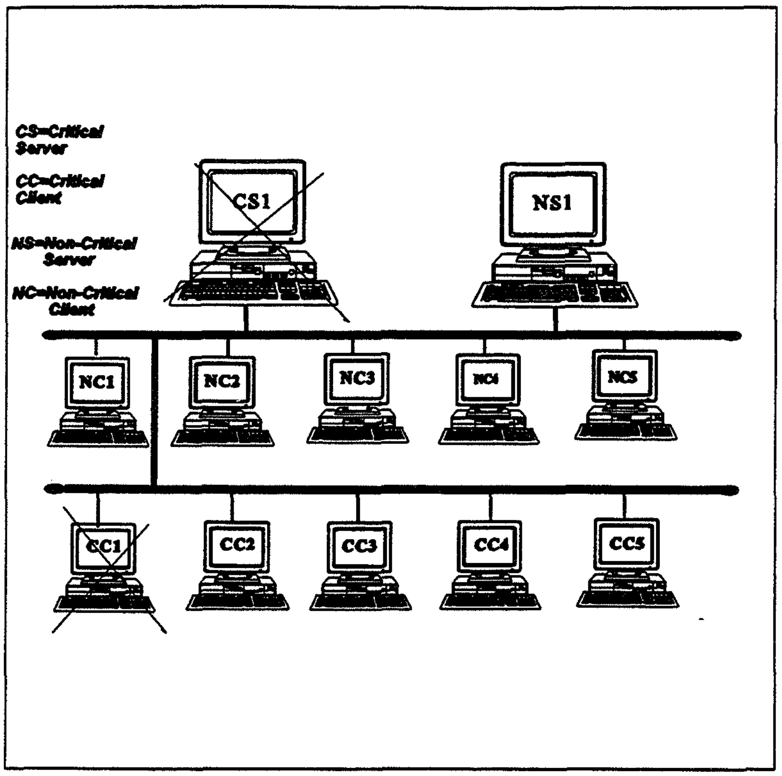


Figure 1. Surviving Configuration

Model Concepts

o The model concepts are illustrated in Figures 1 and 2, where there are five critical clients, five non-critical clients, one critical server, and one non-critical server.

o Figure 1 shows a surviving configuration, where a critical client fails and a critical server fails but there *are* non-critical clients and a non-critical server to take over the functions of the failed nodes.
- The consequence of this configuration is a *Degraded - Type 2* failure mode.

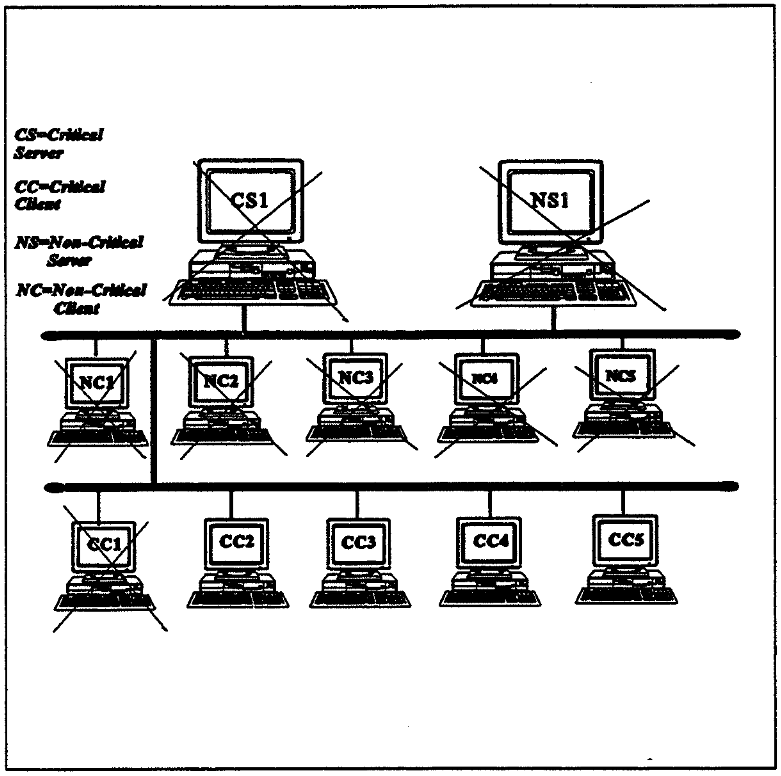


Figure 2. Failing Configuration

o Figure 2 shows a failing configuration where there are *no* non-critical clients and server to take over for the failed nodes.

  - The consequence of this configuration is a *system failure*.

Time to Failure Prediction

In order to make *Time to Failure* predictions for each of the four types of node failures, we first analyze the defect data to determine what type of software defects could cause each of the four types of node failures; then we partition the defect data accordingly. Next we apply the time to failure equation of the *Schneidewind Software Reliability Model* to make each of the four predictions, using the *SMERFS* software reliability tool.

In the equation, $T_f(t)$ is the predicted time (intervals) until the next F_t failures (one or more) occur, α and β are failure rate parameters, s is the first interval where the observed failure data is used, t is the current interval, and $X_{s,t}$ is the cumulative number of failures observed in the range s,t.

$$T_F(t)=[(\log[\alpha/(\alpha-\beta(X_{s,t}+F_t)])/\beta]-(t-s+1)$$

$$\text{for}\quad(\alpha/\beta)>(X_{s,t}+F_t)$$

Time to Failure Prediction (Continued)

Time to Failure predictions are made for critical clients, non-critical clients, critical servers, and non-critical servers.

As the predicted failure times are recorded, we observe whether the condition for system failure has been met. If this is the case, a predicted system failure is recorded. Thus, in addition to monitoring the types of predicted failures (e.g., critical client), the process also involves monitoring $N_{nc}(t)$ and $N_{ns}(t)$ to identify the time t when either is reduced to zero, signifying that the supply of non-critical clients or non-critical servers has been exhausted. In this situation, a failure of a critical client or critical server, respectively, will result in a system failure.

Thus we predict a system failure when the following expression is true:

$((\text{Predict critical client failure}) \wedge (N_{nc}(t)=0)) \vee ((\text{Predict critical server failure}) \wedge (N_{ns}(t)=0))$.

If our predictions produce multiple node failures in the same interval (e.g., critical client and critical server), we record multiple failures for that interval.

APPLICATION OF THE MODEL

o We apply the model to the Marine Corps LOGAIS system -- a client-server logistical support system. It is important that the reliability specification distinguish between failure states *Degraded-Type 1*, *Degraded-Type 2*, and *System Failure*, as previously defined (i.e., distinguish between node failures that cause performance degradation but allow the system to survive, and node failures that cause a system failure). We make this distinction when analyzing the system's defect data.

The defect data used in the example are from LOGAIS defect data [HEI96, MHB96, MTP96]. We use the configurations in Figures 1 and 2.

o In this Windows-based client-server system, we use the classes of clients and servers previously defined, with corresponding classes of defects and failures, as identified in the defect database, and the following short-hand notation for identifying the attributes of the defect database:

o S: Software Defect

o G: General Protection Fault

o N: Network Related Defect

o D: System Crash

Node Failure Counts

o The LOGAIS defect database was queried in order to identify the software defects that qualify as node failures. The following Boolean expressions, corresponding to the four types of node failures, were used:

1. Critical Client Failure: COUNT as failures WHERE $(S \wedge G \wedge N \wedge \mathit{not}C)$. A *GPF* causes a node failure (*Degraded-Type 2*) on a critical client, a client which must maintain communication with other nodes on the network (*Network Mode*), and the failure does not cause a *System Crash* (loss of server).

2. Non-Critical Client Failure: COUNT as failures WHERE $(S \wedge G \wedge \mathit{not}N \wedge \mathit{not}C)$. A *GPF* causes a node failure (*Degraded-Type 1*) on a non-critical client, a client which does not have to maintain communication with other nodes on the network (*Standalone Mode*), and the failure does not cause a *System Crash* (loss of server).

Node Failure Counts (Continued)

3. Critical Server Failure: COUNT as failures WHERE (S∧*not*G∧N∧C). A *System Crash* causes a node failure (*Degraded-Type 2*) on a critical server, a server which must maintain communication with other nodes on the network (*Network Mode*), and the failure is not a *GPF;* it is more serious, resulting in the loss of a server.

4. Non-Critical Server Failure: COUNT as failures WHERE (S∧*not*G∧*not*N∧C). A *System Crash* causes a node failure (*Degraded-Type 1*) on a non-critical server, a server which does not have to maintain communication with other nodes on the network, and the failure is not a *GPF;* it is more serious, resulting in the loss of a server.

o The above classification associates *GPF* with clients and *System Crashes* with servers; it also associates *Network Related Failures* with critical node failures.

Defect Database

An **example** of the defect database is shown in Table 2, where *Interval* identifies the period for counting defects (daily in this case), *Defect ID* is the identification assigned the defect, *Number* is the count of defects in the interval, *Submit* is the date the defect was submitted to the defect database, and the last four columns indicate whether the defects resulted in one of the four types of failure.

Upon querying the defect database, using Boolean expressions 1-4, we find the failure counts listed in the sample database in Table 2. The failure counts corresponding to types 1, 2, 3, and 4 above are summarized in Table 3, which shows the empirical probabilities of node failure.

Table 2

Defect Database (Sample)

CC: Critical Client Node Failure
NC: Non-Critical Client Node Failure
CS: Critical Server Node Failure
NS: Non-Critical Server Node Failure

Interval	Defect ID	Number	Submit	CC	NC	CS	NS
51	2633,2634	2	1/24/95		X		
51	2635,2636,2637,2638	4	1/24/95				X
52							
53	2661,2662,2663,2664	4	1/26/95		X		
54	2641,2644,2645,2669, 2671,2672,2673,3003	8	1/27/95		X		
54	2640,2643,2670,2674, 2675,2676,2783	7	1/27/95				X
55	2450	1	1/30/95		X		
56							
57							
58	2487	1	2/2/95				X
59	2511,2512,2513	3	2/3/95		X		
60							
61	3025,3026,3027,3029	4	2/7/95	X			

Table 3

Summary of Node Failures (4048 Software Defects)

	Number of Failures	Probability
1. Critical Client	24	p_{cc}=.005929
2.Non-Critical Client	83	p_{nc}=.020250
3.Critical Server	2	p_{cs}=.000494
4. Non-Critical Server	158	p_{ns}=.039032
ɔ. Total	267	p_{sw}=.065705

CONCLUSIONS

o Based on the above approach, it is feasible to develop a system software reliability model for a client-server system.

o In order to implement the approach, it is necessary to partition the defects and failures into classes that are then associated with critical and non-critical clients and servers.

o Once this is done, predictions are made of *Time to Failure* for each class; the predictions are classified according to those that would result in a software failure and those that would result in a system failure; and the probability of system failure is computed.

o It is important that software failures not be treated as the equivalent of system failures because to do so would grossly understate system reliability.

o Possible model enhancements include the following: extend the model to include hardware failures; develop measures of performance degradation, as nodes fail; include a node repair rate to reflect the possibility of recovering failed nodes during the operation of the system.

S73-61
41668
360742
P40

Operational Test Readiness Assessment of an Air Force Software System: A Case Study

Amrit L. Goel, Syracuse University
Capt. Brian Hermann, AFOTEC, NM
Major Randy McCanne, Scott AFB, IL

This paper describes a new methodology that was developed to assess operational test readiness of an Air Force software system under development over a period of several years. The evaluation is primarily based on an analysis of the open and closed problem reports. Other factors such as test completeness and requirements stability are also considered, but mostly in an implicit way. The methodology is objective, has a sound mathematical foundation and can be employed for evaluation of any large software system.

AFOTEC Software Maturity Evaluation Guide provides details of the data needs and assessment approach to be used for Air Force Systems. The key criterion is to determine whether the unresolved severity 1 and 2 failures can be resolved prior the scheduled OT&E (Operational Test and Evaluation) start date. The approach taken in the AFOTEC Guide is to estimate the time required for resolution based on the current unresolved failures and an average closure rate. This methodology extends and builds upon the current AFOTEC approach by (i) considering the as yet undetected faults in the system, and (ii) using two different estimated fault closure rates.

An equivalent problem in commercial applications is to determine readiness for beta test, readiness for release, or readiness for first customer ship. Several studies over the past twenty years have attempted to address this problem for both defense and commercial systems. Most of these have proposed using a decision rule in conjunction with some software failure model to predict software readiness. Others have proposed approaches based on minimizing a predefined cost function.

The new methodology employs statistical trend tests and software reliability models for assessing readiness for dedicated OT&E. It explicitly incorporates the use of these techniques in the decision making process by employing an iterative three step procedure:

Step 1. Perform statistical trend analysis

Step 2. Select software reliability model that best fits the system failure data

Step 3. Conduct a readiness assessment using the results of Steps 1 and 2

The basic idea behind the proposed methodology is to determine objective readiness information from the available data on problem reports and their closures. In addition to studying the plots of cumulative open and closed problems and average time to close, it uses a statistical assessment of the trends in the failures and closed fault curves. Each of the steps has a solid mathematical foundation.

In particular, it uses the Laplace trend statistic for determining whether the software failure rate is steady, improving, or deteriorating. When an improving trend is indicated, the failure process is modeled by an appropriate software reliability model.

Information from the trend plots is used to guide model selection as well as to obtain initial model parameter estimates. Next, the reliability model is used to estimate the future failure detection pattern. An initial assessment of OT readiness is then made by accounting for the number of failures remaining open, the problem closure rate and the expected new failures likely to be detected. This is done using the reliability model selected, the actual number of open problems and a stochastic model fitted to the problem closure curve. Such assessments are made for four different cases. Information from the Laplace trend plots and other factors such as rate of testing can also be used to decide whether earlier data should or should not be considered for modeling and readiness assessment.

The presentation will address the following topics:

- Proposed methodology
- Laplace trend test and its relationship to software reliability models
- Description of development data from an Air Force system
- Analyses of open and closed problems, and readiness assessment
- Limitations and benefits of the methodology.

The methodology described here is currently being used on other commercial and defense systems. This paper will provide an assessment of the experience gained and problems encountered in these applications. Suggestions for improvements and any progress on them will also be discussed.

Keywords/Phrases:

- Software Change Trends
- Software Defect Density
- Software Immaturity Case Study and Lessons Learned
- Software Maturity
- Software Operational Testing
- Software Problem Trends
- Software Test Readiness

Operational Test Readiness Assessment of an Air Force System: A Case Study PART 1

Brian G. Hermann
Captain, United States Air Force

Air Force Operational Test and Evaluation Center
Software Analysis Division

1. Introduction

Prior to purchasing space, aircraft, or communications systems, the Air Force operationally tests them to ensure they meet the specified needs of their users. The Air Force Operational Test and Evaluation Center (AFOTEC) conducts these operational tests for the Air Force. Since many modern systems rely heavily on software, the Air Force requires software to be mature before beginning these lengthy, expensive tests.

Software maturity is a measure of the software's progress toward meeting documented user requirements. The software analysis division at AFOTEC uses software problem, change, and failure tracking data to help demonstrate when software has sufficiently met requirements and fixed identified problems. The concept and evaluation are simple, but rarely considered by developers and acquirers prior to AFOTEC involvement.

AFOTEC evaluates software maturity with three distinct goals:

1. **Test Readiness**	Reduce tax dollars wasted on testing immature systems.
2. **Readiness for Fielding**	Determine how far the software has progressed toward satisfying user needs.
3. **Identify Software Maturity Drivers**	Identify portions of the software system which currently generate the most changes and may, therefore, be expected to generate the largest future maintenance effort. Where possible this information should be used to improve the software prior to fielding.

Table 1: Software Maturity Evaluation Goals

2. Evaluation Background

2.1 Software Maturity Data and Collection

The evaluation begins with the software maturity database. Many programs use different names, but the required data is almost always collected by development organizations. Collection and analysis of the data typically begin when the software is placed under formal configuration control and continues through fielding of the software. The minimum data required to evaluate software maturity is shown in Table 2.

1. Software Change (Problem) Number
2. Description
3. Computer Software Configuration Item (CSCI) Identifier
4. Severity Level
5. Date Change Opened (or problem found)
6. Date Change (Problem) Closed and Implemented

Table 2: Software Maturity Data

During the development and initial testing, developers and acquirers work together to assign a severity level rating to each problem. Later during operational testing, AFOTEC is responsible for scoring of software problems. The Air Force uses a standard five-point scale shown in Table 3.

2.2 Severity Level Categorization

Current Air Force policy requires that no system can progress to the operational testing phase with open severity level one or two software problems. According to these definitions, severity level one and two problems imply the system does not meet user needs and therefore operational testing would be a waste of time and money.

Severity Level	Description
1	Mission failure or jeopardizes safety.
2	Mission degraded with no possible work-around.
3	Mission degraded but a work-around solution is known.
4	Operator inconvenience or annoyance
5	Any other change.

Table 3: Software Problem Severity Levels (MIL Standard 498)

2.3 Weighting of Severity Levels

To help estimate the operational impact of each change, we assign a weight to each severity level (Table 4). The description of the trend charts will show how these weightings can help to distinguish between many insignificant problems and many important problems.

Severity Level	Weight (Change Points)
1	30
2	15
3	5
4	2
5	1

Table 4: Weighting Factors

2.4 Maturity Evaluation and Analysis Tool

AFOTEC developed a Microsoft® Excel for Windows™ based tool, called Maturity Evaluation and Analysis Tool (MEAT), to automate the data manipulation, produce trend charts, and speed analysis and reporting. The tool and user's manual are available at no cost from HQ AFOTEC/SAS.

2.5 Evaluation Indenture Level

While software maturity can be evaluated at the system software level, it is also beneficial to look at maturity from lower indenture levels. Selecting the appropriate evaluation indenture level is based on software size, number of changes, and the length of time the software change data is collected. As a general rule, we suggest the software maturity should be evaluated to at least the CSCI level. For some large programs, it will be possible and beneficial to delve deeper to the computer software component (CSC) indenture level. In either case, the results help to determine which components or configuration items are causing maturity problems. This specific information helps the acquiring organization and the developer more effectively address problems.

2.6 Synthesis of Many Trends

Software maturity is not a single trend or evaluation. It is a synthesis of many trends that must be considered together with the external factors that influence them.

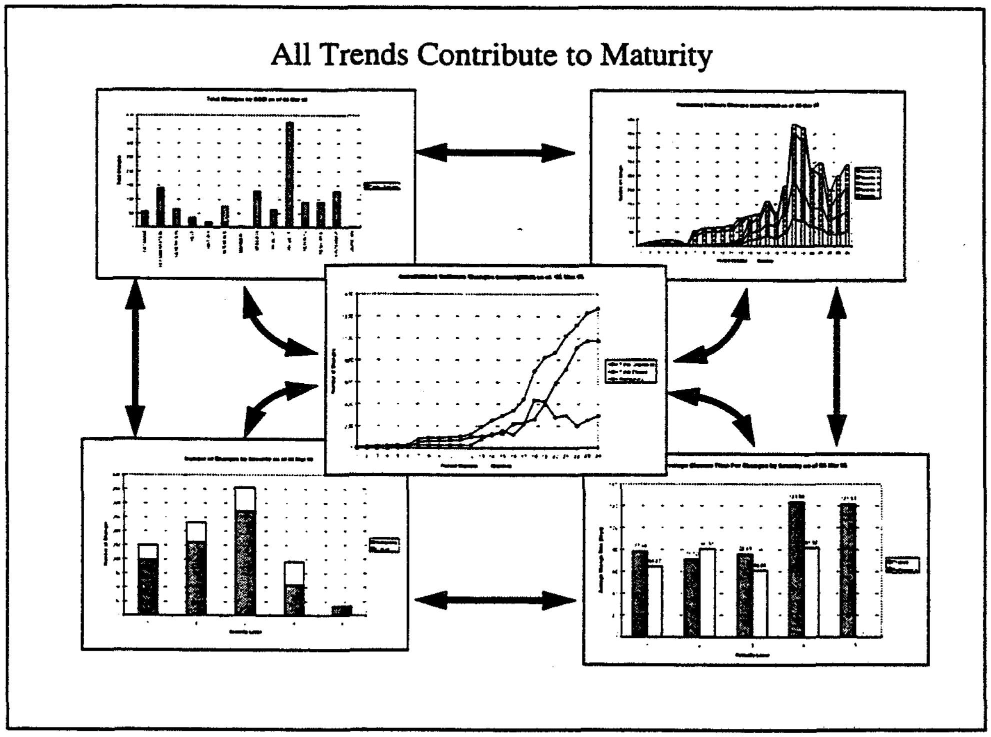


Figure 1: Software Maturity – A Synthesis of Many Trends

2.7 External Factors

2.7.1 Test Rate

One of the external factors that can affect software maturity is developmental test schedule. This aspect can be seen in both test rate and test completeness. An understanding of test rate helps the evaluator determine if software appears mature only because testing has slowed, or explain an unusually high change origination rate resulting from an aggressive test schedule. The test rate should, in fact, affect the slope of the *total originated changes curve*. A sample test rate chart is shown in Figure 2.

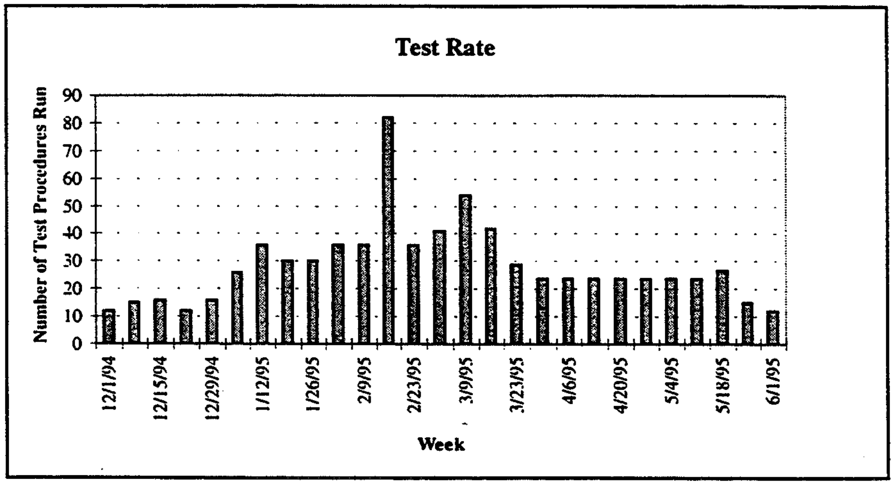


Figure 2: Test Rate by Week

2.7.2 Test Completeness

Another way program schedule can affect software maturity is through test completeness. This measure enables the evaluator to estimate confidence in the software maturity evaluation. A high percentage of successfully completed test procedures, with respect to the total number of test procedures, indicates testing has identified a correspondingly high percentage of problems. One drawback to this measure is that traceability between test procedures and requirements or functions is not part of test completeness, but it is necessary to verify the thoroughness of testing. Figure 3 shows an example of test completeness. Notice the total number of test procedures typically increases during the development and testing.

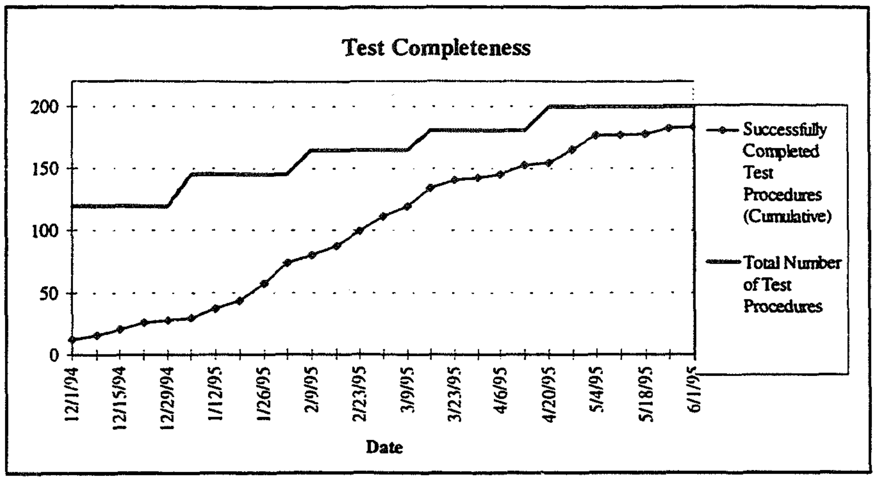


Figure 3: Test Completeness

2.7.3 Requirements Stability

Another factor which influences software maturity trends is requirements stability. Software requirements continue to grow and change in nearly every development. New or modified requirements will likely drive software changes and increase the slope of the *total originated changes* curve. Knowing the cause for software changes can help to pinpoint solutions.

3. Trend Charts

3.1 Weighted and Unweighted Software Changes

This basic maturity chart (Figure 4) shows the *total changes originated*, *closed*, and *remaining* trends. This chart is also a good example the ideal shape of each trend line. To indicate maturity or progress toward maturity, the *total changes originated* trend should begin to level off. This indicates testing is finding problems at a lower rate than earlier in the development and testing. The *total changes closed* curve should closely follow the identified changes. Ideally, all identified changes would be closed and the *remaining changes* curve would show no backlog. This chart is also presented in an unweighted form as well as individually for each severity level.

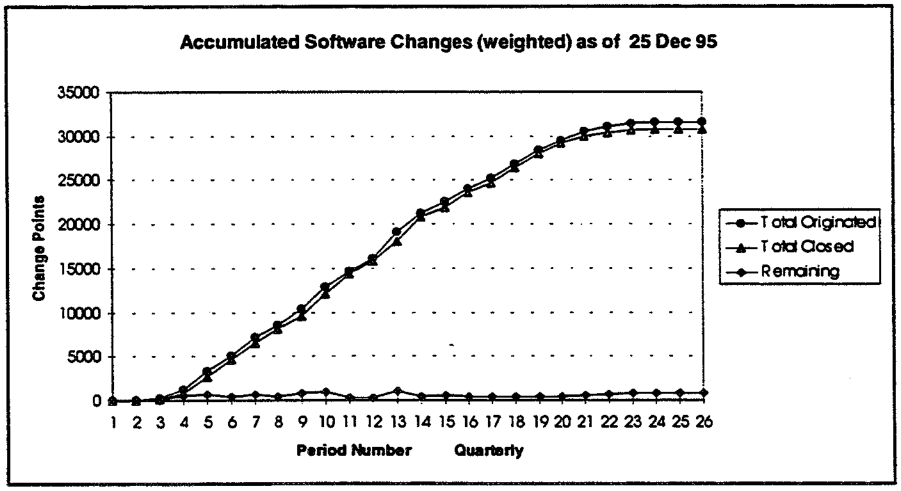


Figure 4: Accumulated Software Changes (Weighted)

3.2 Remaining Problems

Although the *remaining changes* trend in an unweighted chart shows the current software problem/change backlog, Figure 5 presents a more useful view. This stacked bar chart shows the overall backlog trend as well as each severity level's contribution to the total backlog.

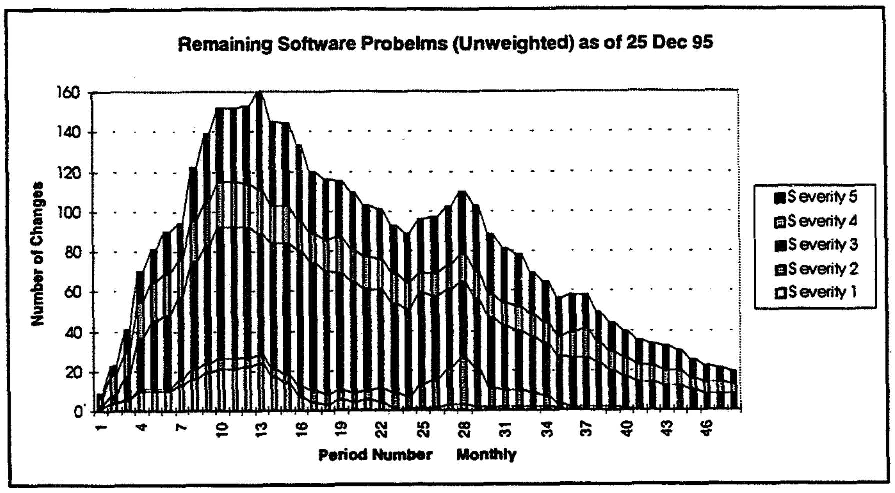


Figure 5: Remaining Software Problems

3.3 Average Severity Level

In the next chart (Figure 6), we present the average severity level of all originated, closed, and remaining changes. Ideally, the average severity level of problems should drop over time. Another good sign is if remaining changes are of a lower average severity level than those changes already closed. This indicates that the developer is doing a good job prioritizing his efforts.

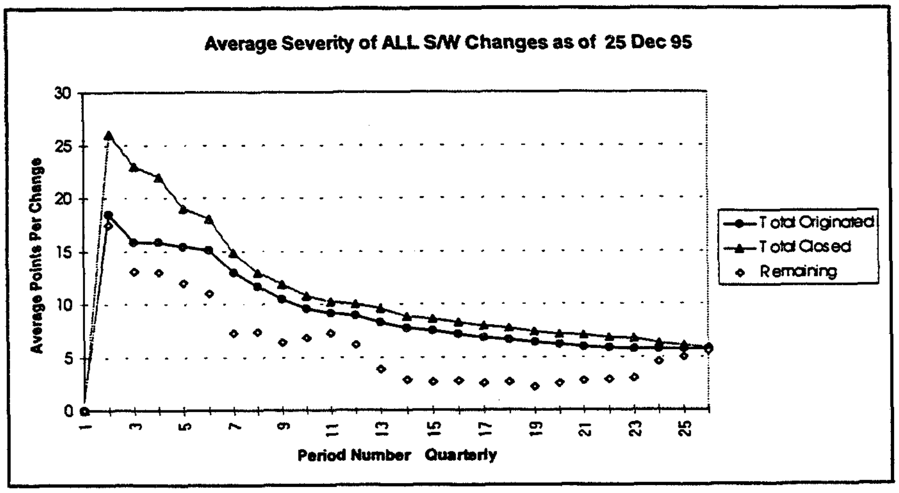


Figure 6: Average Severity Level

3.4 Distribution of Changes by Severity Level

Although Figure 7 is not actually a trend, it shows how the changes are distributed by severity level. The sample chart exaggerates the expectation that most changes will be of lower severity level.

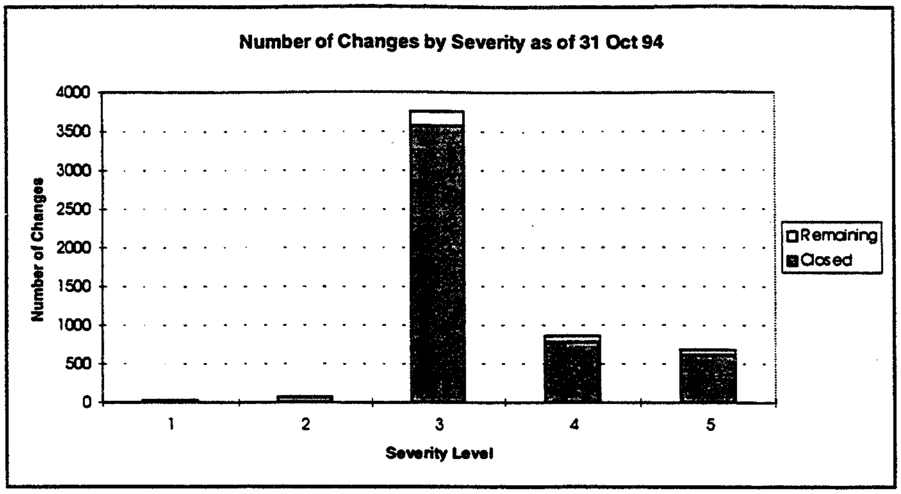


Figure 7: Distribution of Changes by Severity Level

3.5 Average Closure Time by Severity Level

Figure 8 shows the average length of time required to close problems and change requests of each severity level and the average length of time that remaining changes have been open. Understanding this information and the process used to implement changes helps to estimate much change traffic to expect, how many software maintainers will be required, and how far away the software is from being ready for release.

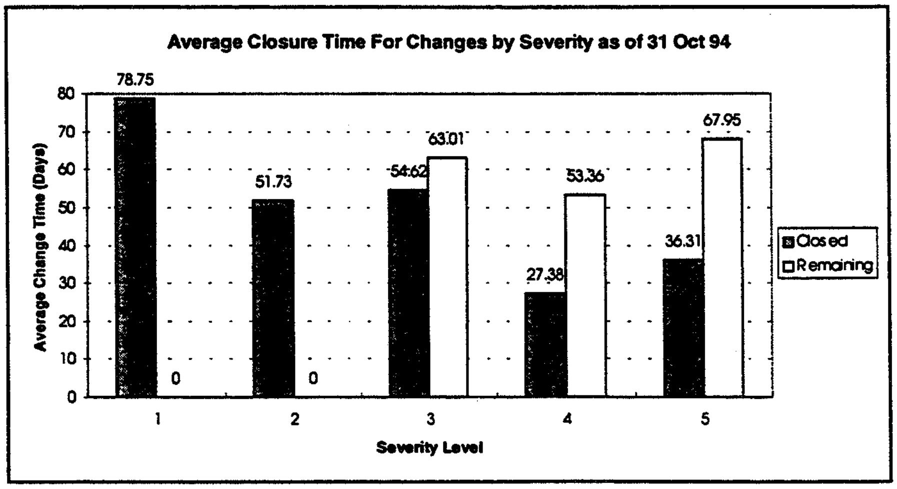


Figure 8: Average Closure Time

3.6 Total Changes and Change Density

The total number of changes for each CSCI helps to identify software maturity problem areas. In addition to sheer numbers of changes, normalizing changes by the size (new or modified lines of code) for each CSCI shows which parts of the code have the most change requests and are most likely to require future effort. We call this normalized measure, *change density*.

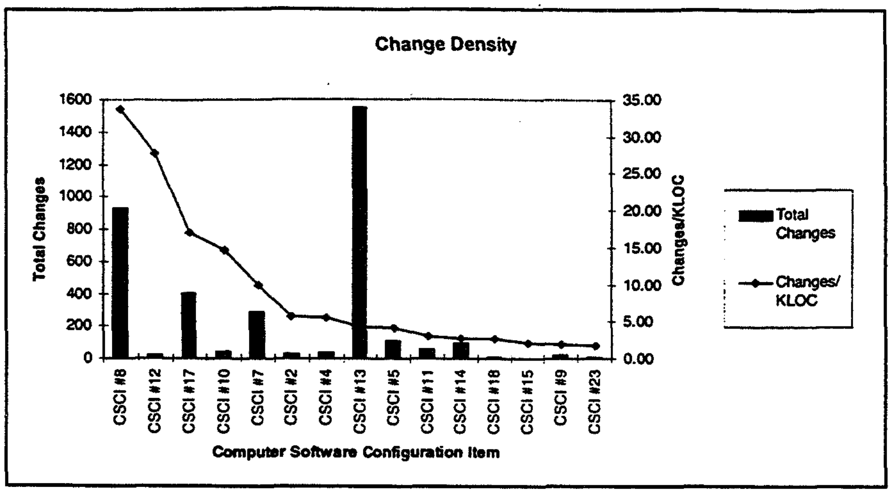


Figure 9: Total Changes and Change Density

From the bars in Figure 9, we identify CSCIs #8, #17, #7, and #13 as portions of the software which have produced large numbers of changes. The lines on the same chart identify CSCIs #8, #12, #17, #10, and #7 as components which produce large numbers of changes per line of code. The union of these two sets can be thought of as maturity drivers for the software system

3.7 Remaining Changes and Defect Density

The final trend chart is a relatively new addition to our evaluation methodology Figure 10 shows both remaining changes for each CSCI and the number of remaining changes (problems) divided by thousands of new or modified source lines of code (defect density). Michael Foody suggests software is not ready for release until the defect density is below 0.5[1]. Finding portions of software with the most remaining problems and the highest defect densities are two additional pieces to the maturity puzzle

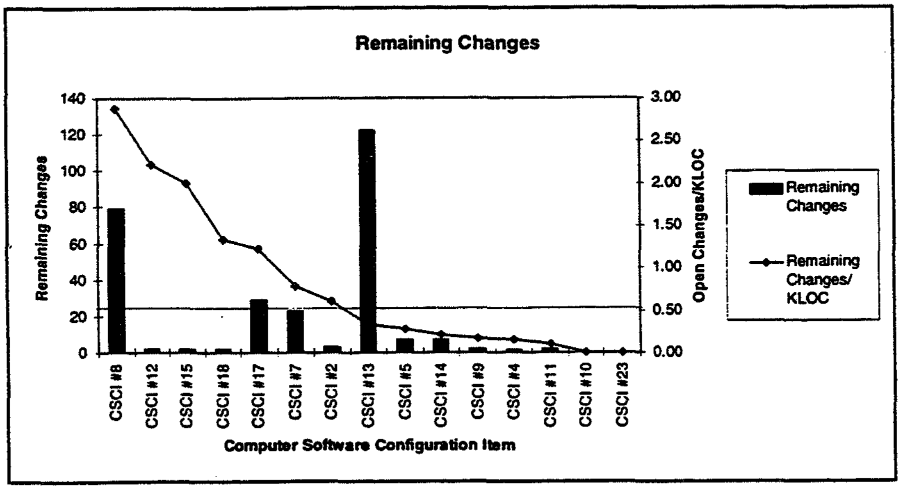


Figure 10: Remaining Changes and Defect Density

The bars in Figure 10 identify CSCIs # 8, #17, #7, and #13 as components with large numbers of remaining changes. The CSCIs with a defect density above the 0.5 threshold (CSCIs #8, #12, #15, #18, #17, #7, and #2) are not ready for operational testing or release. The union of these two sets are the software components which are currently driving software immaturity.

[1] Michael A. Foody, "When is Software Ready For Release?" *UNIX Review* March 1995

4. Case Study

This section is a time-phased example of software maturity evaluation for a major Air Force acquisition program. The program was selected because it's initial immaturity presents a convincing case for delaying operational use of software until maturity. The program name and developer will not be identified.

4.1 Initial Evaluation - March 1995

The initial evaluation of the software maturity was analyzed and briefed in March of 1995. Although the data was available to, and in fact from, the development organization, software maturity was not evaluated except to track open software change requests. The acquiring organization understood there were problems in the development, but had not evidence of how severe the problems were or where the problems were located.

4.1.1 Weighted and Unweighted Software Changes

Like most software developments, problems were initially found much more quickly than they were being fixed. Unfortunately, this trend continued up to the point of our initial evaluation. As shown in Figure 11, the *total originated* and *total closed* trends diverge except for a push to close changes from week 16 through 20. The result is an increasing backlog of software changes. At that time, we had no reason to expect a slowdown in change origination and current closure rates do not predict improvement.

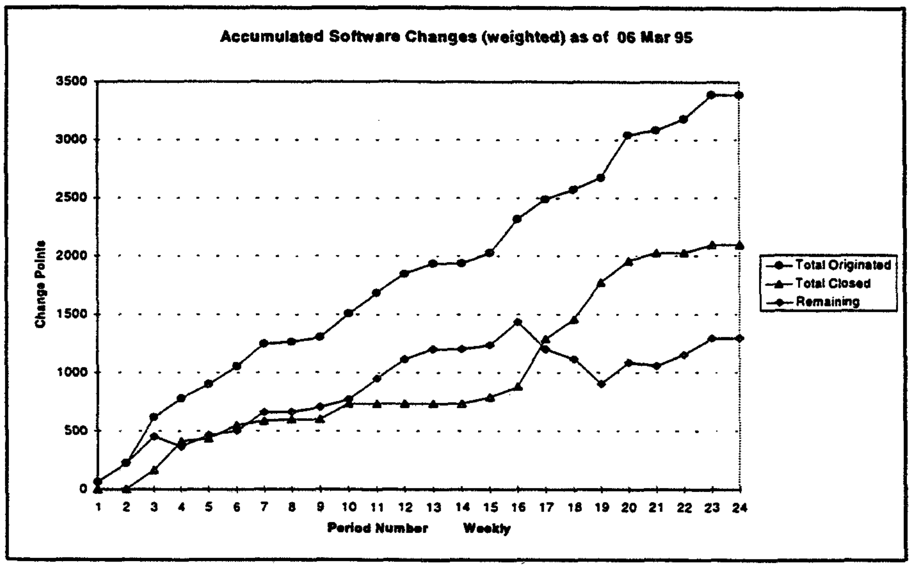


Figure 11: Initial Evaluation Weighted Changes

The unweighted version of this chart (Figure 12) looks almost identical. The only difference is that the numbers in this chart represent actual changes and backlog size rather than the change points used in the weighted chart.

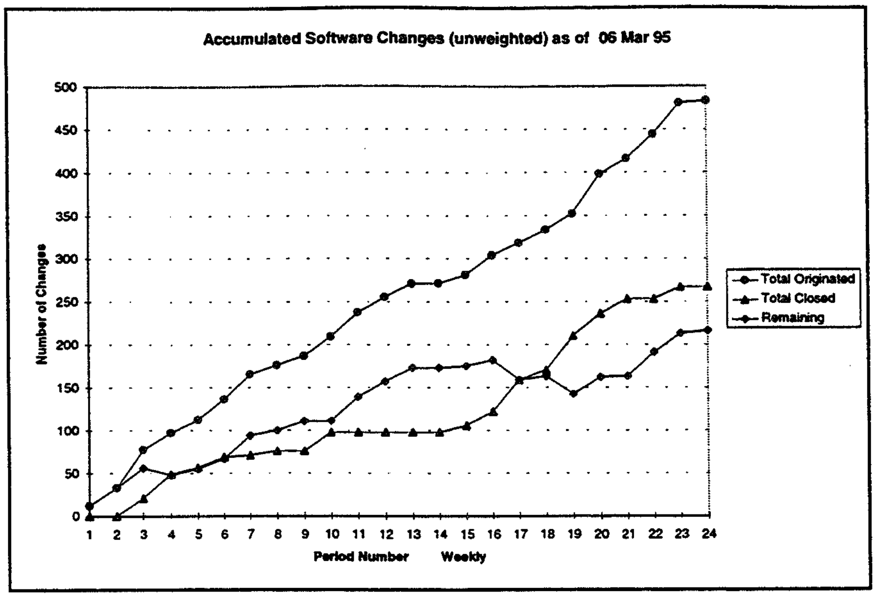


Figure 12: Initial Evaluation - Unweighted Changes

4.1.2 Average Severity Level

The average weight of changes (problems) throughout the period up to the initial evaluation was between six and eight (Figure 13). This equates to between a severity level two or three change. The only positive trend shown by this chart is that the developer has recently been working on the most severe problems.

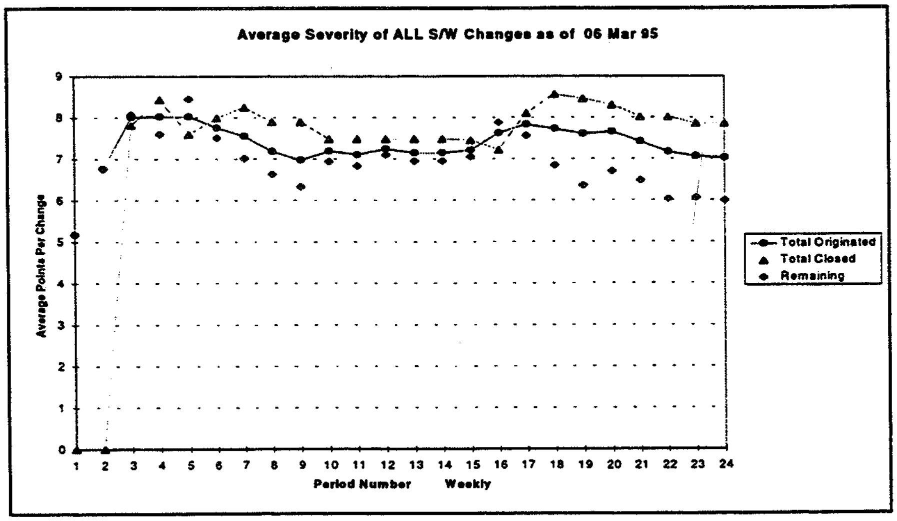


Figure 13: Initial Evaluation - Average Severity Level

4.1.3 Distribution of Changes by Severity Level

The distribution of changes across severity levels showed two surprising results. First, an unusually large number of severity level one changes were opened and remained open. Second, very few severity level two changes had been identified. Overall, this chart spurred a discussion of severity level definitions.

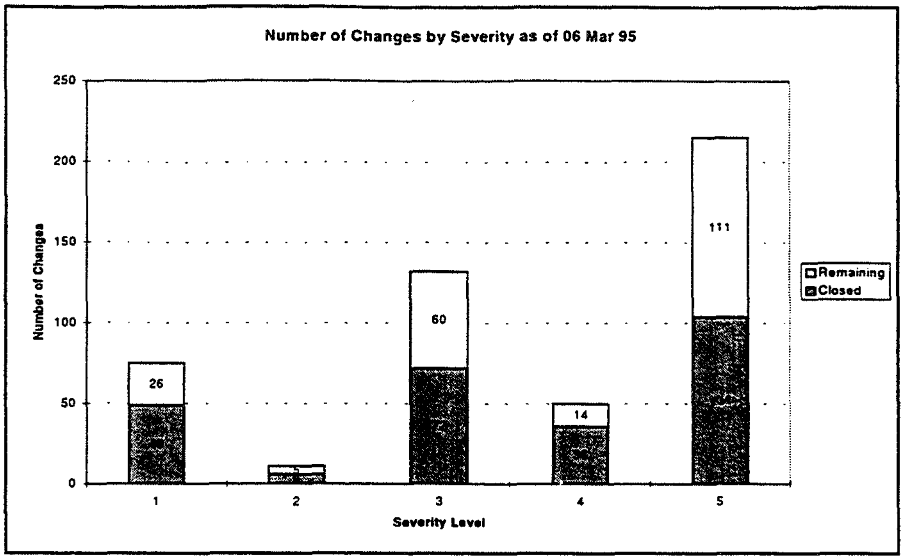


Figure 14: Initial Evaluation - Number of Changes by Severity Level

4.1.4 Average Closure Times by Severity Level

The next set of trends (Figure 15) showed that most problems had historically taken between 35 and 40 days to formally close. Unfortunately, changes that were currently open at that time had, with the exception of severity level five, been open longer than the average of those already closed. This indicates that closure times will likely rise in the future. Because difficulty and severity level are not synonymous, we were careful not to compare closure times across severity level.

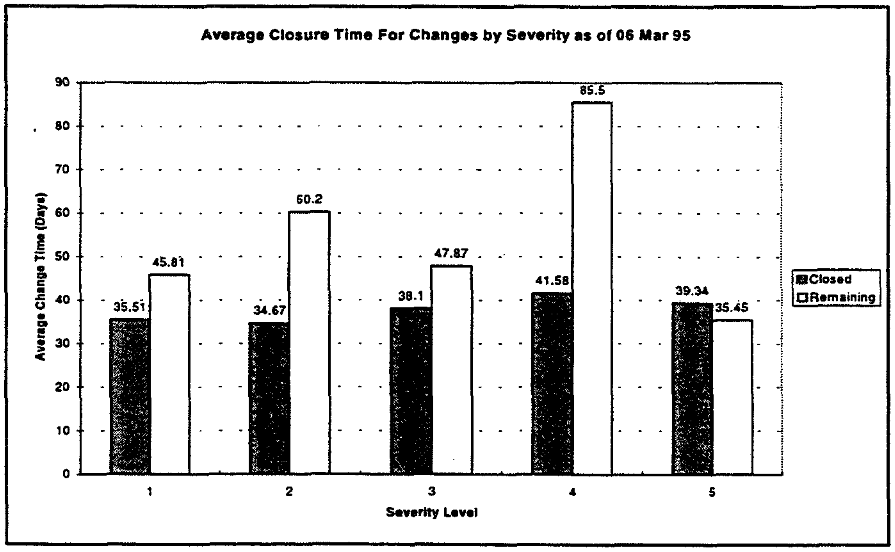


Figure 15: Initial Evaluation - Average Closure Times

Since the maturity data for this development program did not include information about which portions of the code the changes/problems related to, we were unable to produce change and defect density charts.

4.1.5 Summary

Nearly all of the trends pointed to immaturity of the software. In addition, we knew the test schedule was consistently being shortened to save time at the tail-end of the development. All parties agreed to further study this data on a biweekly basis until the test readiness decision in early August 1995.

4.2 Test Readiness Decision Evaluation - July 1995

Between the initial evaluation and the test readiness decision, the developer modified severity levels of many of the problems to reflect a better understanding of the severity level definitions. As a result, software maturity was not as bad a previously thought.

4.2.1 Weighted and Unweighted Software Changes

A great deal of progress was made toward closing the backlog (Figure 16, Figure 17, and Figure 18). Notice that changes in the slope of the curves are more dramatically shown on the weighted chart. For example, between weeks 13 and 17 on Figure 16, we see a great deal of progress in closing problems. The trend is more dramatic on the weighted chart because the problems were of high severity levels.

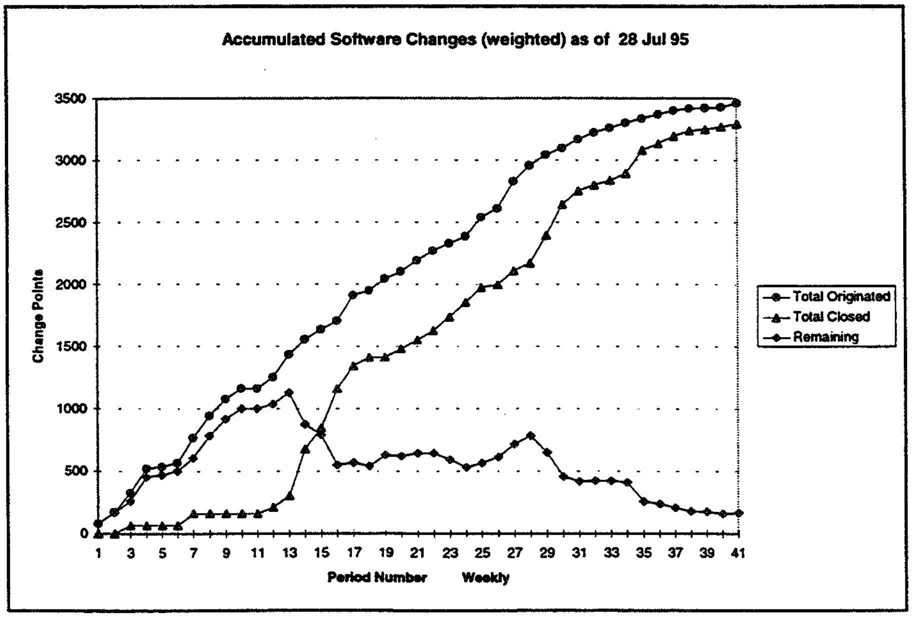


Figure 16: Test Readiness - Weighted Software Change Trends

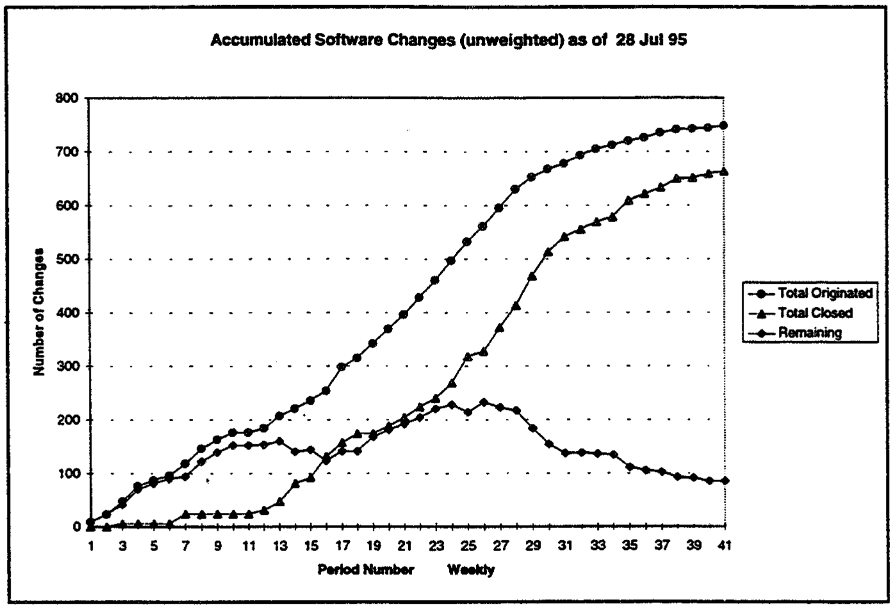


Figure 17: Test Readiness - Unweighted Software Change Trends

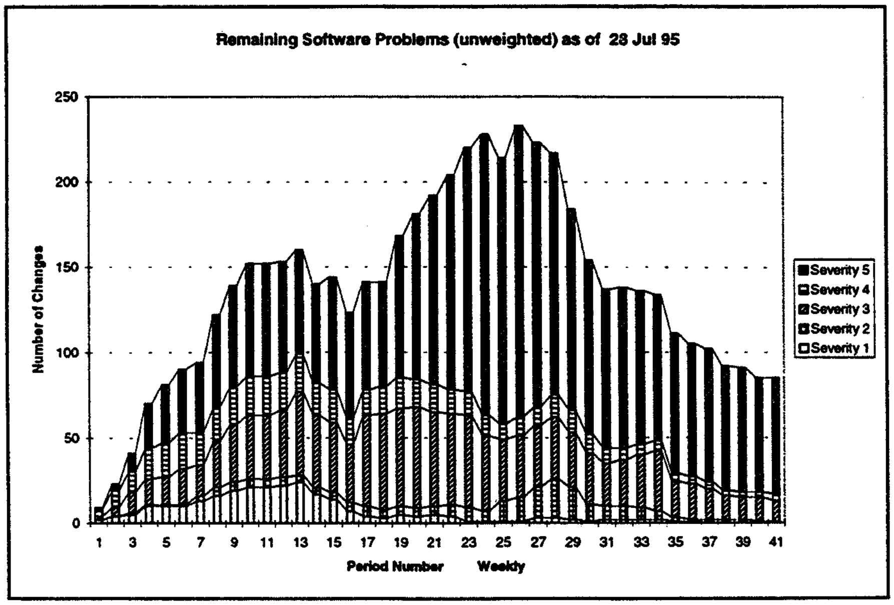


Figure 18: Test Readiness - Remaining Software Changes

4.2.2 Average Severity Level

Figure 19 shows the average severity level of recently opened, closed, and remaining changes has decreased and remained stable for the last three months.

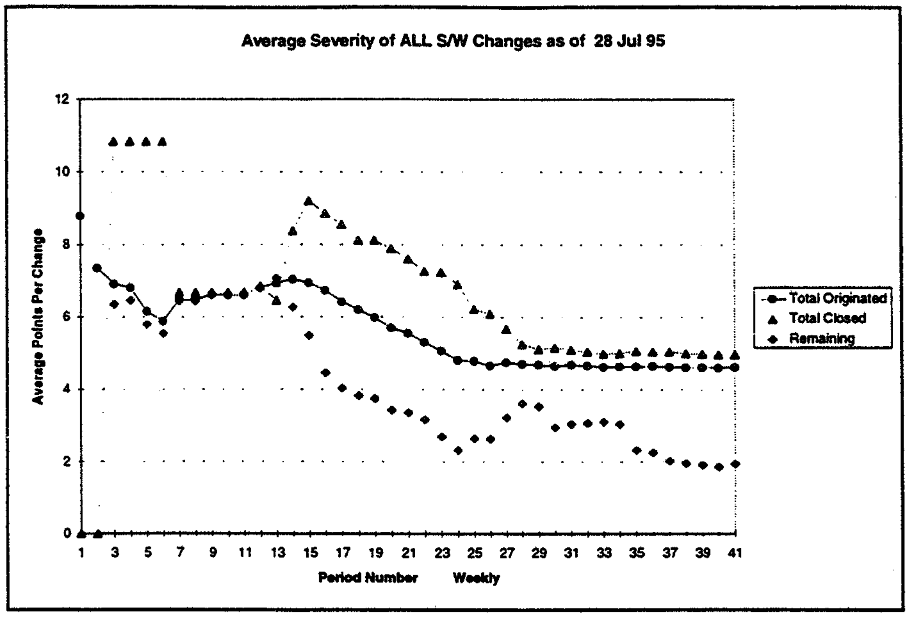


Figure 19: Test Readiness.- Average Severity Level

4.2.3 Distribution of Changes by Severity Level

The developer's better understanding of severity level definitions resulted in a distribution of changes that is closer to normal expectations. Unfortunately, one severity level one change remains unresolved. This means that execution of some part of the software will result in a mission failure or jeopardize safety.

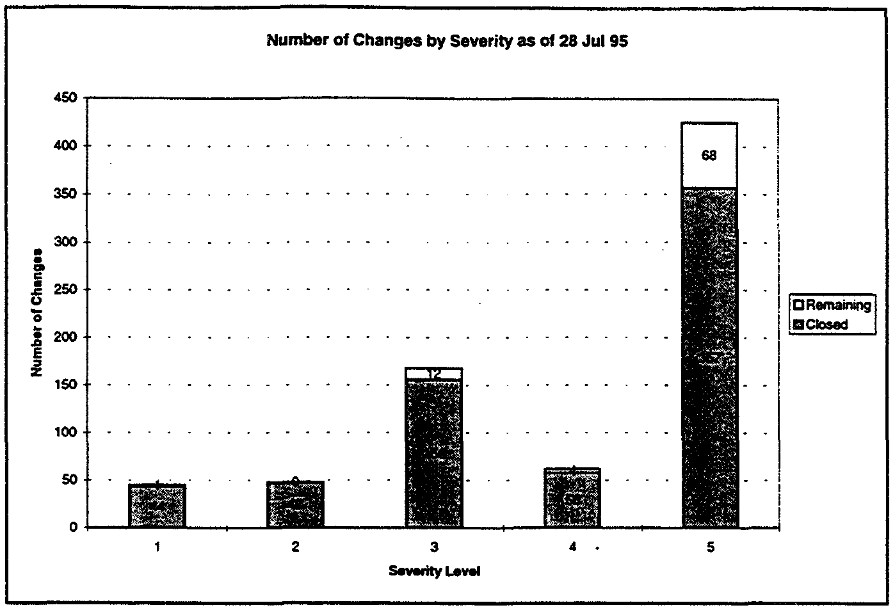


Figure 20: Test Readiness - Change Distribution

4.2.4 Summary

The software showed signs of improving maturity, but local trends were too short to absolutely declare the software mature. For this reason and because of the open severity level one problem and the reduced testing schedule, we declare the software not ready for test.

Due to schedule and funding constraints, the system proceeded to the operational testing phase despite maturity problems. Although this decision did not follow recommendations, we were anxious to see how the results matched with our maturity analyses to date.

4.3 Initial Operational Use - August 1995

Just days before the first operational test exercise of the software, a new version was delivered and checked out on the system. During the first familiarization session of the software for field operators rather than system developers, the software worked less than 40% of the time. This list of work around procedures to software problems grew to over 100.

Finally during the first operational use of the software, it failed dramatically. A software failure caused an incomplete safety notification to system users. The system allowed the users to bypass the warning and overheat some sensitive electronic equipment. As a result, the one-of-a-kind system was out of commission for two months, $1.5 million in hardware repairs were required, a new version of software produced and tested, and expensive test time was lost until October 1995.

4.4 Extended Operational Testing - October 1995

After the lengthy delay, the system was once again accepted for test. Largely due to this delay, the software maturity charts appeared mature. Fortunately, this time the operational testing was run to completion. Unfortunately, the system had performance problems as well as user interface troubles. In fact, users stated they would, "prefer to have the old system back." Over 100 software deficiencies were identified during one month of operational use. Six of these software problems were judged to be severity level one and two. Clearly the system, and the software in particular, was not ready for fielding.

4.5 Software Impact on Purchase Decision - March 1996

After a miserable showing during initial operational testing, developers proceeded to fix identified problems prior to the system purchase decision. As a result of preliminary findings, the decision to purchase the system was delayed. The system would undergo a second round of operational testing to look for improvement.

As shown in Figure 21, current maturity trends indicate the software has progressed toward maturity. We must temper this analysis with an understanding that the scope of software testing has been reduced during the period between operational testing and the decision to re-test the system. The impact of this reduced testing is a slower rate of identifying new changes. As a result, the developers were able to fix most of the outstanding changes including all of the severity level one and two changes.

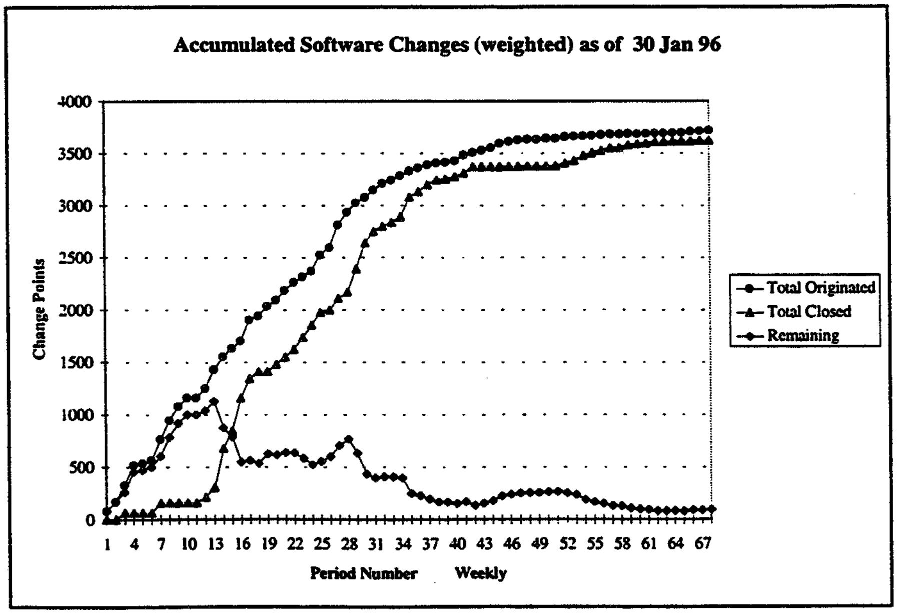


Figure 21: Current Maturity Status

5. Conclusion

Software maturity is a simple evaluation to conduct and interpret, yet the information is extremely useful for developers, acquirers, and operational testers. The trend charts must, however, be interpreted together as a whole and in the context of external factors such as program schedule and requirements stability. As a result, the maturity evaluator must have in-depth knowledge of the software development and testing.

Specifying maturity requirements for release and following through with those decisions will help to ensure time and money are not wasted testing immature software, users are not disappointed with initial software capabilities, and software maintainers receive quality products. In the case study, the software maturity evaluation correctly predicted software immaturity. Failure to listen to this advice resulted in millions of dollars in repair expenses and wasted test time.

PART II

Readiness Analysis for an Air Force System

R. McCanne
Major, United States Air Force
Scott AFB, IL

READINESS ANALYSIS FOR AN AIR FORCE SYSTEM

1. Introduction

Development data from an Air Force software system are analyzed in this section using the three-step methodology. The data consist of weighted originated failures and weighted closed failures. The weights are 30, 15, 5, 3 and 1 for severity levels 1, 2, 3, 4 and 5, respectively. The time period of data is 86 months.

A brief description of the data is given in Section 2. Guided by the trend statistic curve, analyses and maturity assessments are then done at months 70, 75, 80 and 86 in Section 3. A summary of the assessments is presented in Section 4.

2. Data Description

A graph of the cumulative weighted originated failures, cumulative weighted closed failures and weighted failures remaining open is shown in Fig. 21a. These values are called change points and thus the data are cumulative Open Change Points (OCP), cumulative Closed Change Points (CCP) and Remaining Open Change Points (ROCP). A cursory study of the OCP and CCP plots in Fig. 21a indicates very little failure activity for the first twenty-five months. Then there is an almost constant rate of increase up to month 60. This is followed by a convex curve for OCP and an almost straight line for CCP. The ROCP curve seems to be increasing up to month 50 and then remains constant up to month 70. Finally, it shows a decreasing trend up to month 86. A better understanding of their behavior can be gained from the Laplace Trend Statistics curves in Figs. 21b and 22 for OCP and CCP, respectively.

Figure 21b indicates a slight reliability decay and them some growth during the first twenty months. It is followed by stable reliability indication up to month 27, and reliability growth to month 40. Then there are indications of local reliability growth and decay. Starting with month 60, there is strong indication of continuing reliability growth up to the present, viz, month 86. Figure 22 seems to follow a pattern similar to that of Fig. 21b. In practice, analysts track the failure phenomenon and management tries to keep up with the failure curve. In other words, as more change points are originated, management tries to ensure that more are closed.

As mentioned earlier, readiness assessment is a difficult problem. In addition to the open and closed curves, it may require consideration of test rate, test completeness and requirements stability. Since these items are generally not available, the following assessments are based purely on the behavior of the OCP and CCP plots. Reexamining these plots in light of observations made above, it would seem that readiness assessment could have started with month sixty. However, by month seventy, there is strong indication of sustained reliability growth. In the following, the results of assessments at months 70, 75, 80 and 86 are briefly summarized.

3. Assessments at Months 70, 75, 80 and 86

In each case, the Laplace trend statistic curves were studied for total change points, originated and closed. These were used as guides for determining the NHPP model choice and initial parameter estimates as detailed earlier in this paper. After fitting the appropriate models, the best one was selected. The fitted models were then used to estimate the future failure curve and the model closure rate (MCR). The average closure rate (ACR) was computed from the change points remaining open data. The above values were then used to assess readiness. In the analysis given below, the system would be considered ready for release when problems remaining open become zero.

The resulting analyses can be summarized graphically in four figures for each analysis month. The first two figures in each case would show fitted NHPP models to open and closed data, the third problem closure months for cases 1 and 2 and the fourth problem closure months for cases 3 and 4. The figures for each of the analysis months were studied and the results analyzed for readiness assessment. Such plots for months 80 and 86 are shown in Figures 23 to 26 and 27 to 30, respectively.

Assessment Month	70	75	80	86
Case 1	77.4 (332)	84.0 (340)	87.2 (348)	90.4 (349)
Case 2	78.0 (305)	87.2 (254)	90.5 (238)	94.1 (191)
Case 3	83.4 (332)	91.3 (340)	91.5 (348)	92.3 (349)
Case 4	84.9 (305)	98.0 (254)	98.3 (238)	98.8 (191)

4. Summary of Assessments

The above table summarizes the results of various analyses at months 70, 75, 80 and 86. It gives the failure closure month (month all remaining open failures are closed) for each assessment month and for each of the four cases. The corresponding values of ACR and MCR are given in parentheses. Thus for case 1 at month 70, the average failures closure rate is 332 per month and all currently open failures should be resolved by month 77.4. For case 4, month 80, the model based closure rate is 238 per month and current unresolved failures and the failures to be detected should be resolved by month 98.3. A graphical representation of these results is shown in Figure 31.

Some observations from above table are summarized below.

Case 1.

This represents the situation when no new detected failures are assumed and the average closure rate (ACR) is used to close the remaining open problems. For this data set, the ACR is almost constant. The changes in the month to reach zero remaining open problem in each assessment month is due to the additional new failures detected from the previous assessment month.

Case 2.

The model closure rate in this case is decreasing for each successive assessment month because of the decreasing closure rate. It would take longer to resolve the open faults than for case 1 for each respective assessment month.

Case 3.

Compared to case 1 (which also assumes an average closure rate) this case explicitly accounts for the extra time required to resolve the failures to be detected in future months. This is a more realistic situation than case 1 would represent.

Case 4.

Just as in Case 2, the closure rate is decreasing for each successive assessment month. Hence it would take longer to resolve the problems remaining open than in case 3 for each respective assessment month.

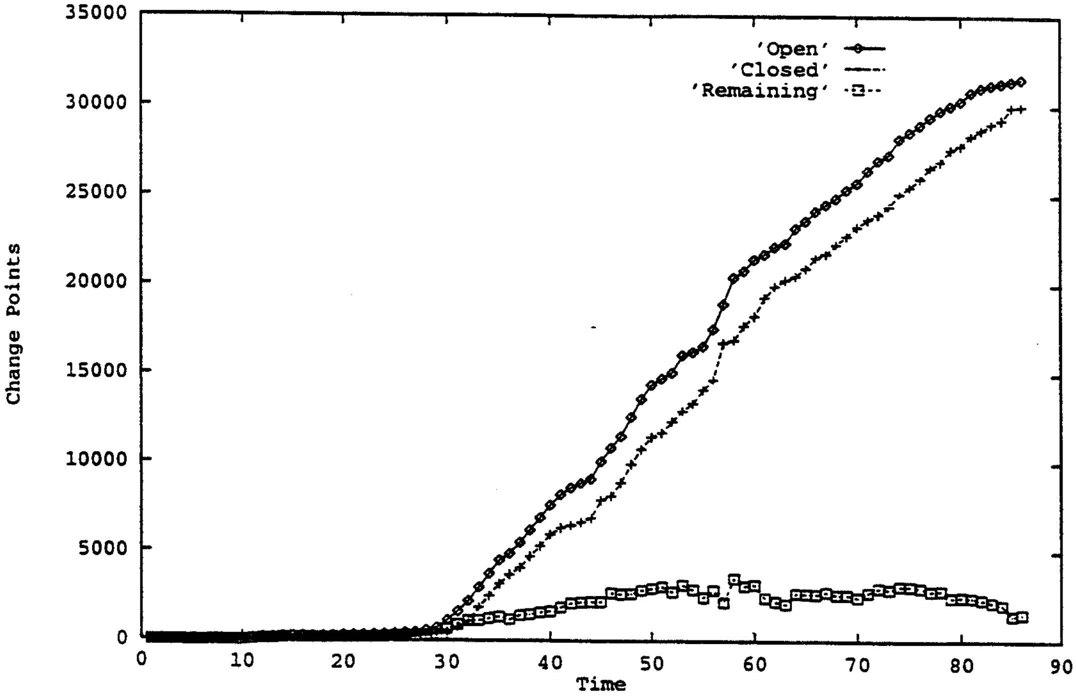


Figure 21a: Accumulated software changes

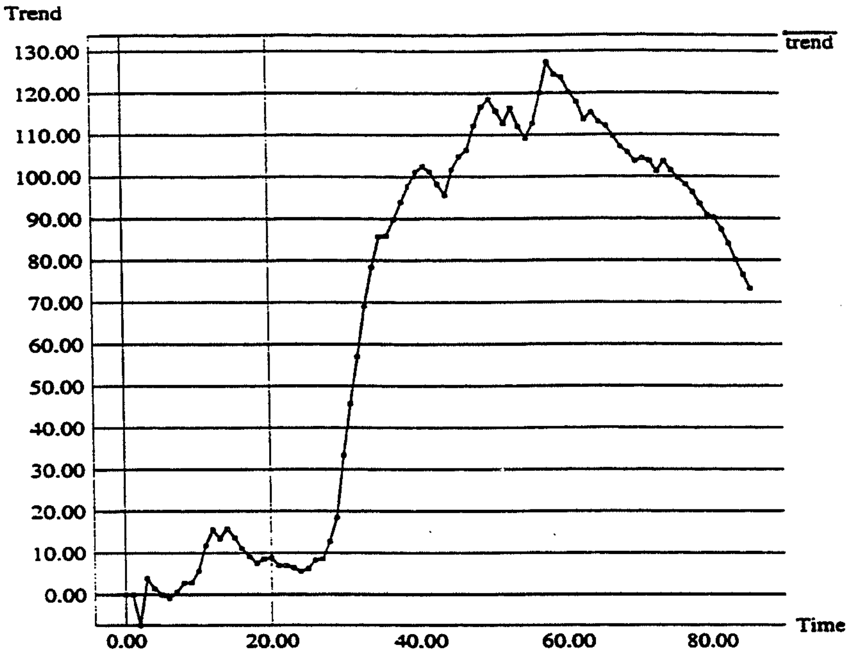


Figure 21b: Trend test for open data

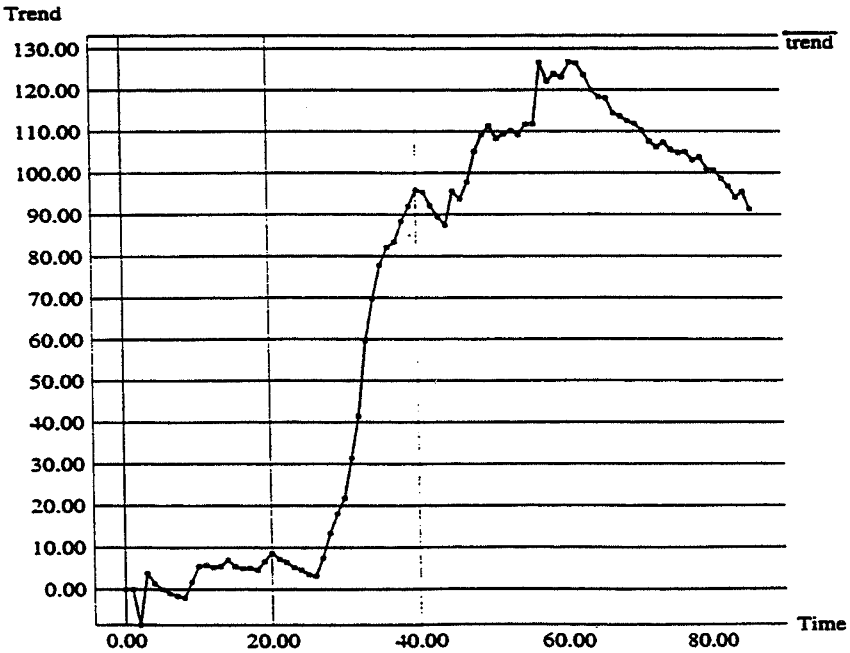


Figure 22: Trend test for closed data

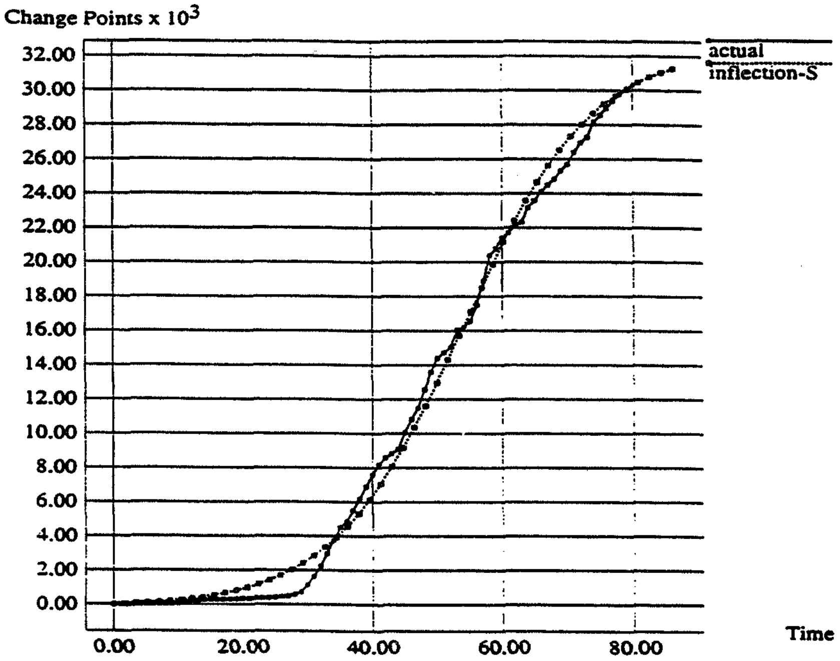


Figure 23: Open data and fitted model at month 80

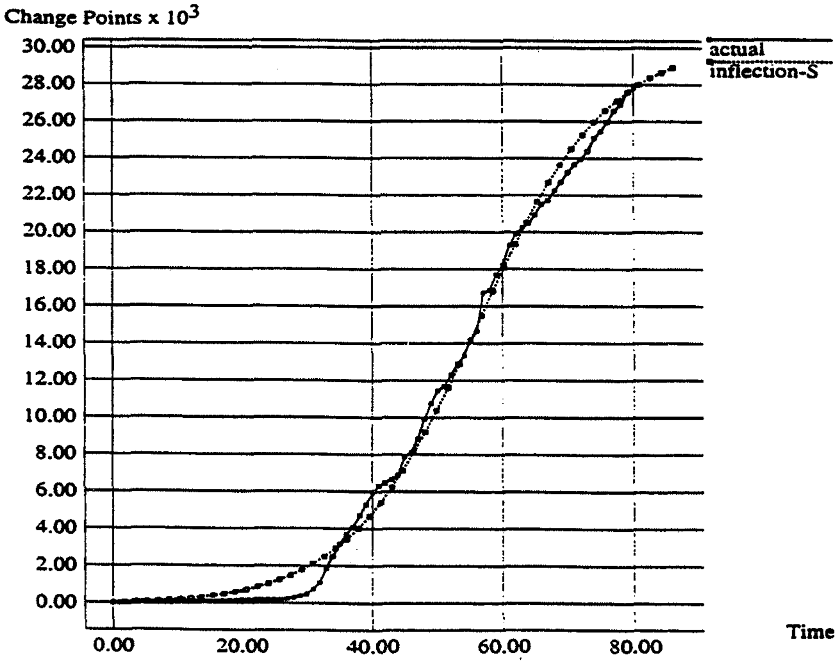


Figure 24: Closed data and fitted model at month 80

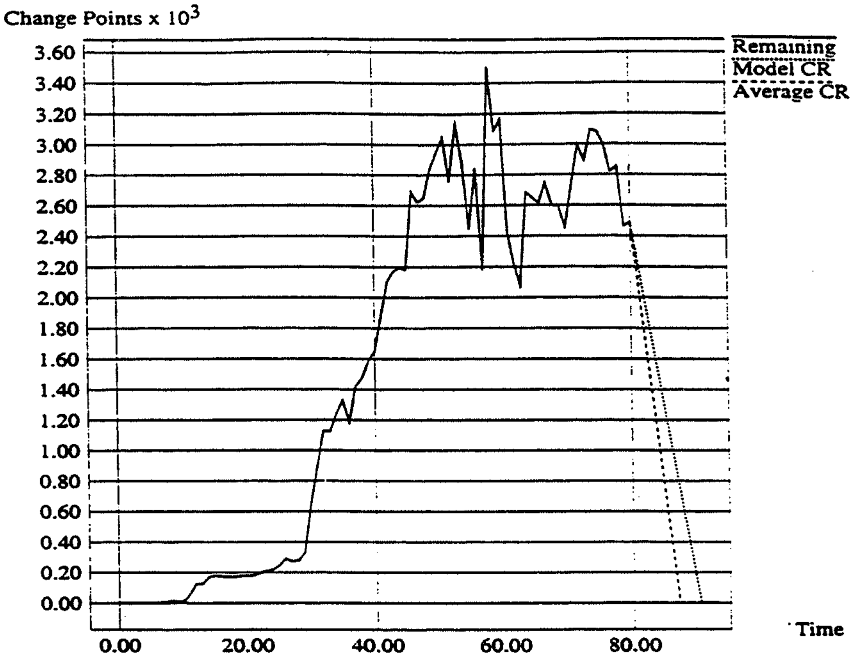


Figure 25: Readiness analysis at month 80 not accounting for new faults

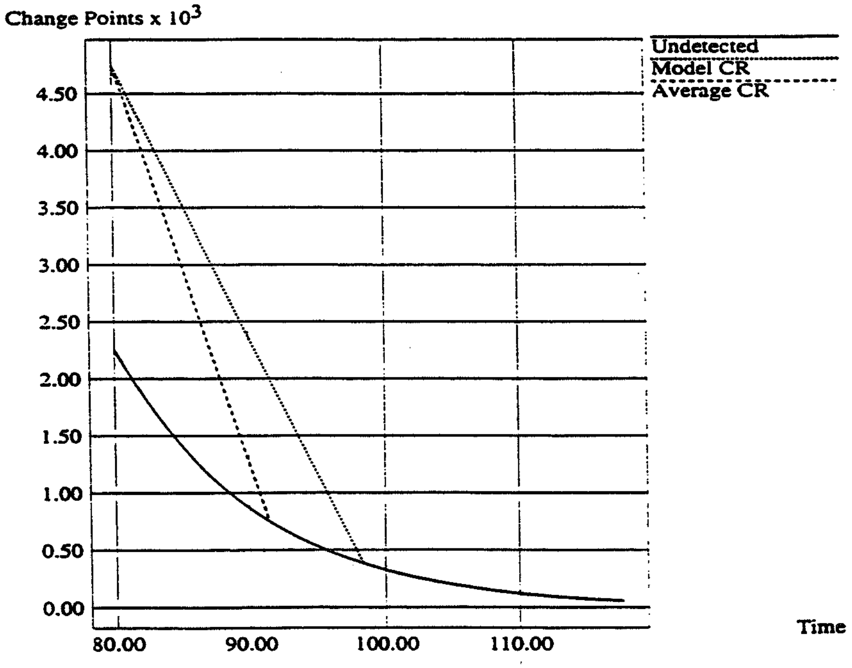


Figure 26: Readiness analysis at month 80 accounting for new faults

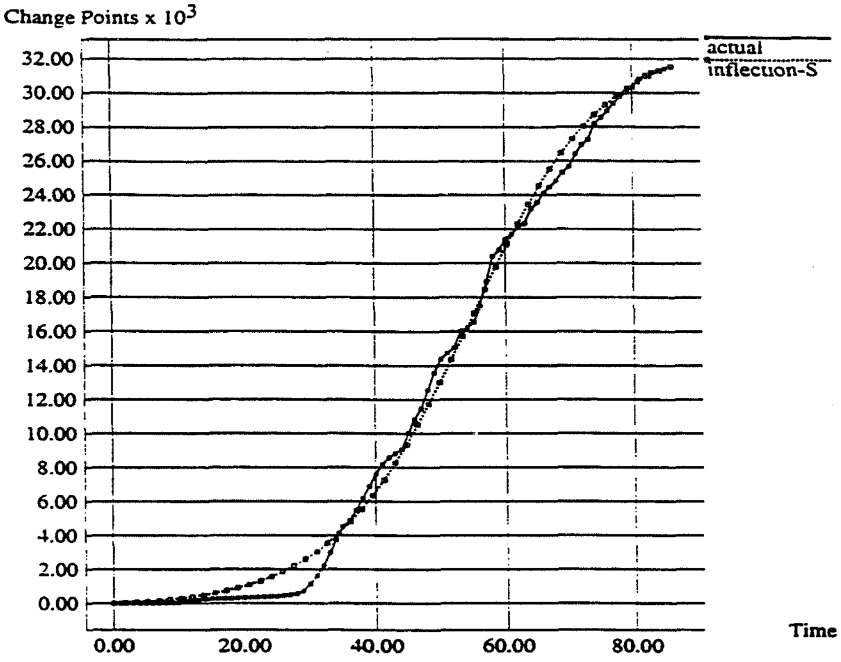


Figure 27: Open data and fitted model at month 86

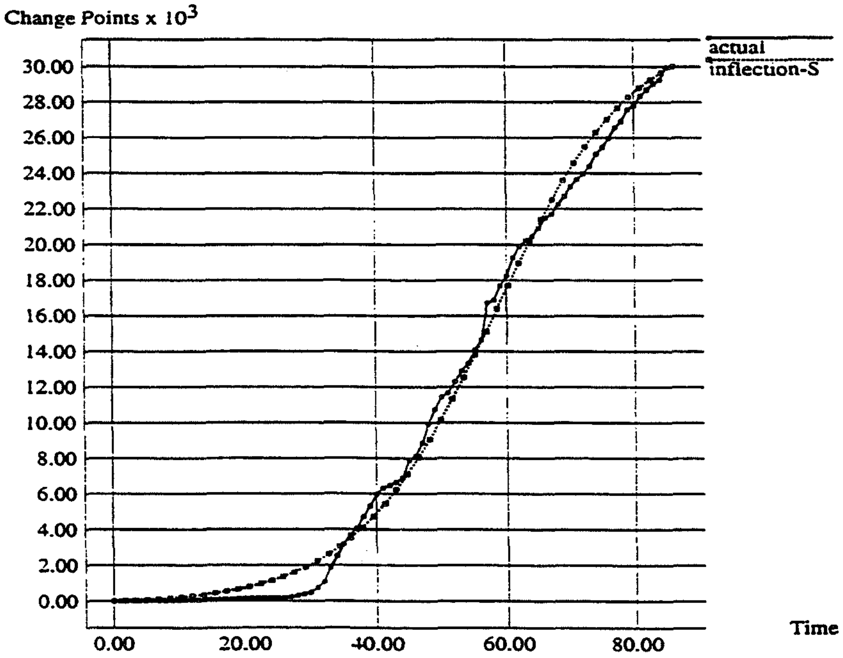


Figure 28: Closed data and fitted model at month 86

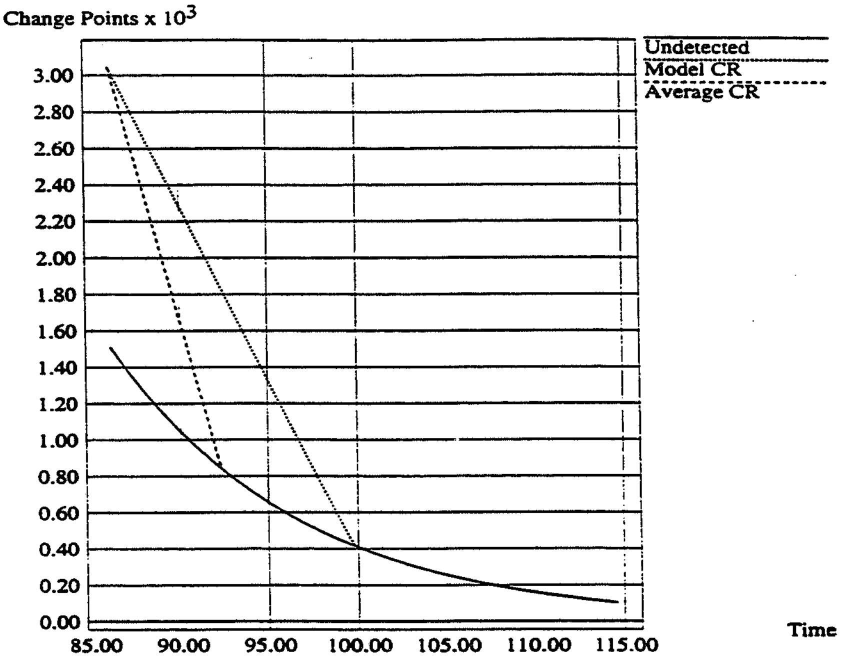


Figure 29: Readiness analysis at month 86 not accounting for new faults

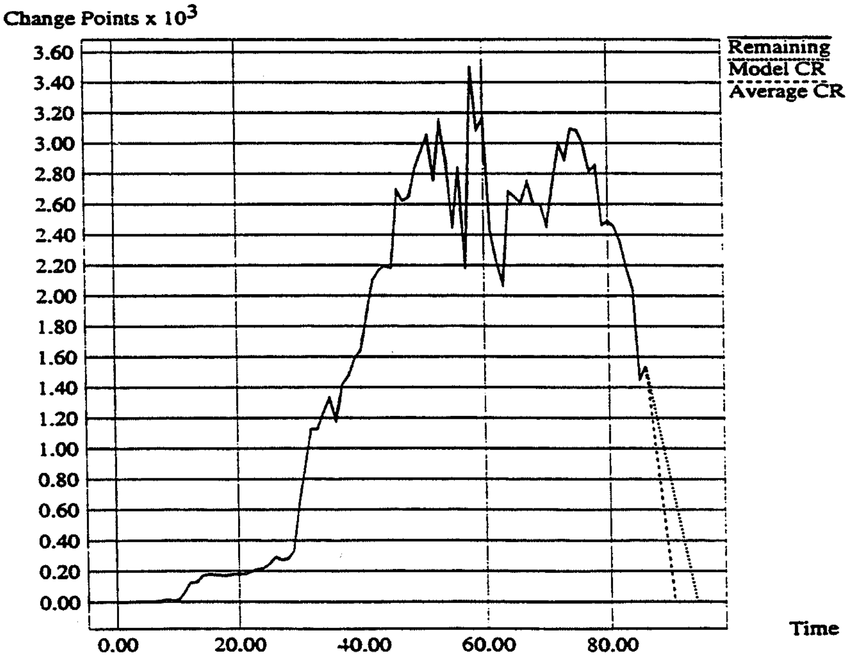


Figure 30: Readiness analysis at month 86 accounting for new faults

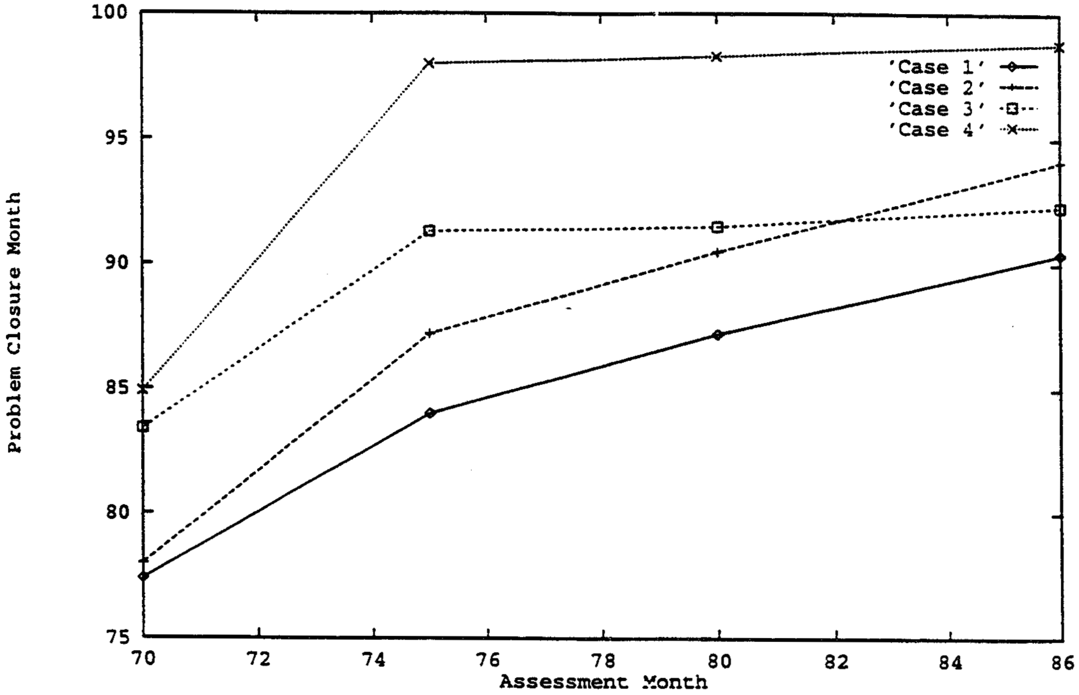
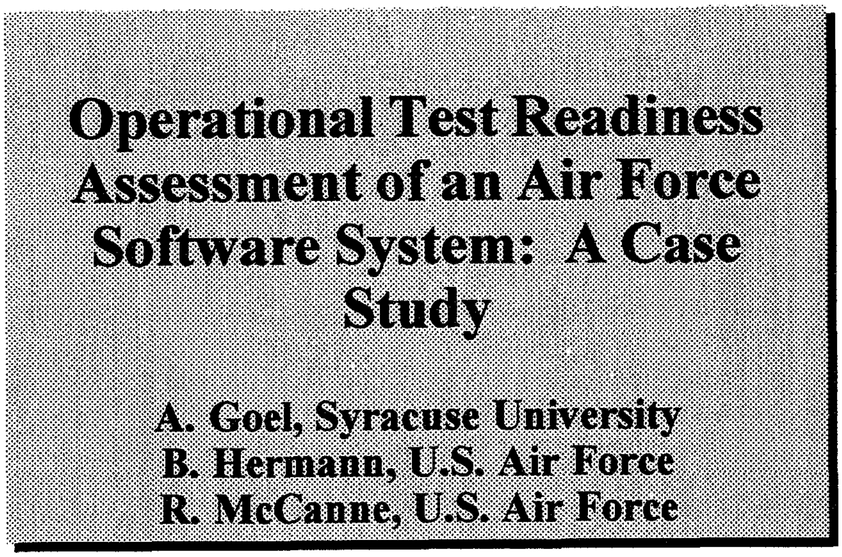
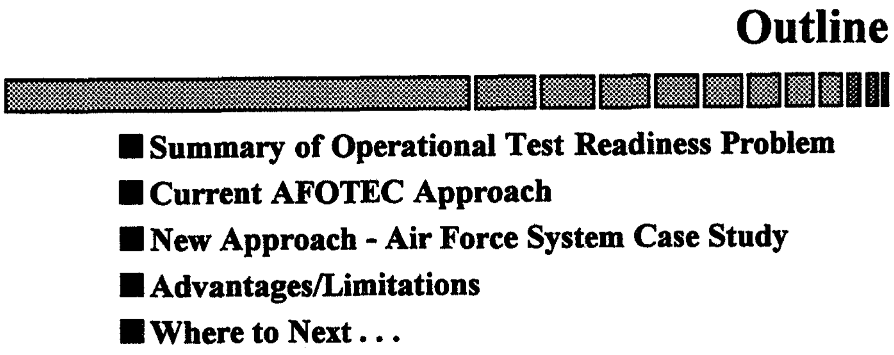


Figure 31: Graphical representation of readiness assessments at months 70, 75, 80 and 86



Operational Test Readiness Assessment of an Air Force Software System: A Case Study

A. Goel, Syracuse University
B. Hermann, U.S. Air Force
R. McCanne, U.S. Air Force



Outline

- Summary of Operational Test Readiness Problem
- Current AFOTEC Approach
- New Approach - Air Force System Case Study
- Advantages/Limitations
- Where to Next . . .

Summary of Operational Test Readiness Problem

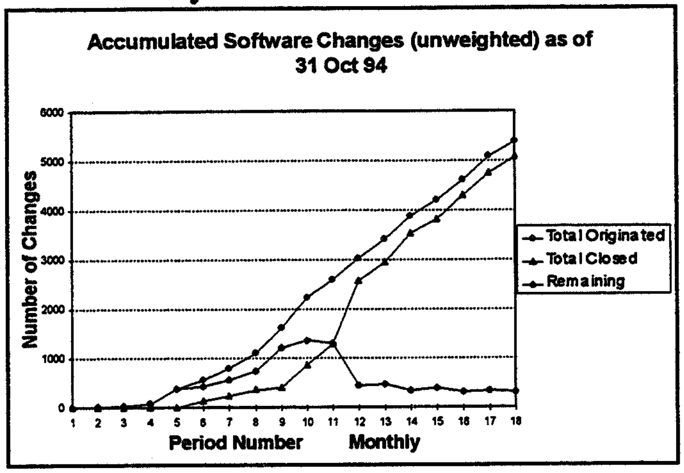
- AFOTEC operationally tests systems to ensure they meet user requirements
- Operational testing is very expensive (especially for embedded systems)
- Need a method for determining if system is ready for operational testing
  - Use templates (checklists)
  - Software maturity evaluation
    - » CURRENT STATUS ONLY
    - » NEED ABILITY TO PREDICT

Current AFOTEC Approach

- Software Maturity - progress software products are making toward meeting user requirements
- Use software problem/change report data and categorize by severity level
  - Change Points = Severity Weight * Number of Changes
- Limited to current status with only crude "straight-line" estimates

Current AFOTEC Approach

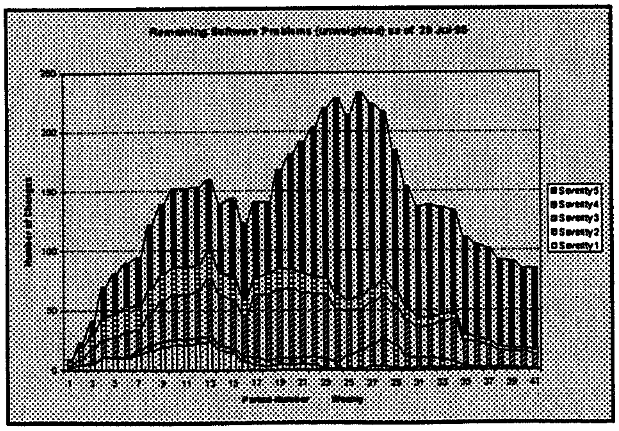
- **Basic Maturity Chart**



Accumulated Software Changes (unweighted) as of 31 Oct 94

Current AFOTEC Approach

- **Backlog Chart**



Remaining Software Problems (unweighted) as of 29 Jul 95

Current AFOTEC Approach

Demonstration of Maturity Evaluation Shortcomings

- **Embedded Air Force System**
- **Completely hopeless initial evaluation**
- **Improving test readiness evaluation - but not yet acceptable**
- **Decision makers proceeded with operational testing**
  - **Found lots of BIG problems -- Required new software build**
  - **$1.5 Million damage to system**
  - **2 Month Test Delay**

Current AFOTEC Approach

Problems With Current Approach

- **Too often ignored by senior decision makers**
- **Lacks the ability to reasonably predict future maturity status**

New Approach - AF Case Study

Three Step Method

1. **Statistical Trend Analysis**
2. **Reliability Growth Modeling**
3. **Readiness Evaluation**

Statistical Trend Analysis

Assess Trends in Data

- **LaPlace Trend (LT) Test**
  - Widely studied, applied to s/w reliability problem
  - UMP unbiased test when paired with some NHPP models
- **Indicates stable, increasing, or decreasing failure rate trend**

Reliability Growth Model

- Used to estimate the impact of undiscovered faults
- Also can be used to estimate the closure rate
- Two common problems . . .
  - Model Selection
  - Estimation of Model Parameters

Readiness Evaluation

Four Cases

- Case 1
  - No new failures
  - Average Closure Rate (ACR)
- Case 2
  - No new failures
  - Model-based Closure Rate (MCR)
- Case 3
  - New failures according to RGM
  - Average Closure Rate (ACR)
- Case 4
  - New failures according to RGM
  - Model-based Closure Rate (MCR)

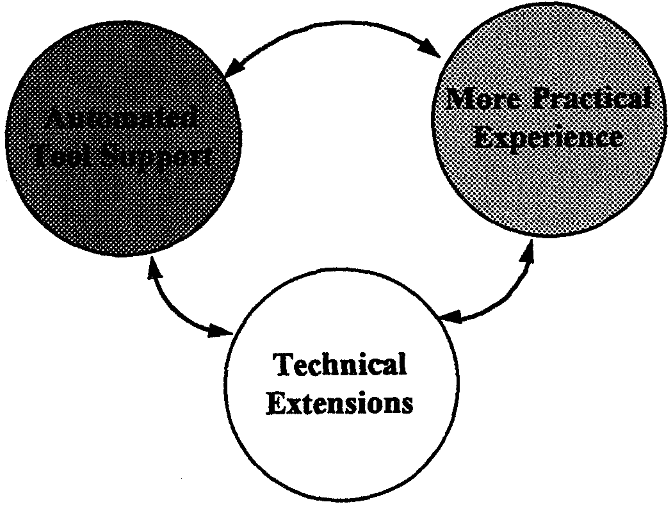
Post-Mortem Results for an Air Force System

	Assessment Month			
	70	75	80	86
Case 1 (ACR)	77.4 (332)	84.0 (340)	87.2 (348)	90.4 (349)
Case 2 (MCR)	78.0 (305)	87.2 (254)	90.5 (238)	94.1 (191)
Case 3 (ACR)	83.4 (332)	91.3 (340)	91.5 (348)	92.3 (349)
Case 4 (MCR)	84.9 (305)	98.0 (254)	98.3 (238)	98.8 (191)

Advantages/Limitations

- **Advantages**
  - **Provides an objective and systematic framework for analytically performing readiness assessments**
  - **Can be adapted to be consistent with current AFOTEC approach**
- **Limitations**
  - **Assumptions must represent actual development environment**
  - **Practical use requires a good understanding of underlying theoretical framework**
  - **Requires tool support to perform necessary analyses**

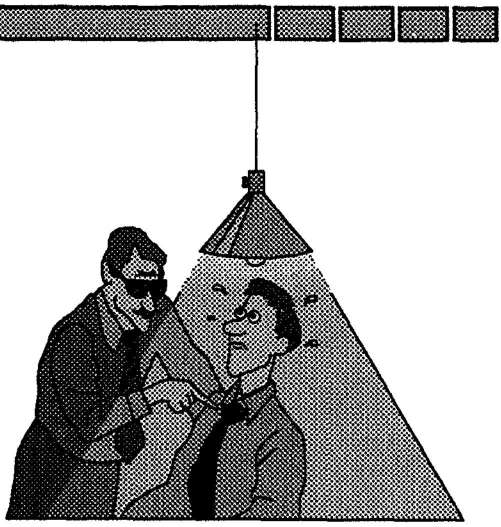
Where to Next . . .



Conclusions

- Proposed a method of three iterative steps for conducting assessments to determine software readiness for dedicated OT&E
- Methodology explicitly uses trend test and reliability models for decision making
- It extends current AFOTEC approach
  - considers undetected faults
  - provides two estimates of fault closure rate

Questions?



Session 5: Case Studies

OMIT THIS PAGE

Risk Knowledge Capture in the Riskit Method
J. Kontio and V. Basili, University of Maryland

Requirement Metrics for Risk Identification
T. Hammer and L. Hyatt, NASA Goddard, W. Wilson, L. Huffman, and L. Rosenberg, Software Assurance Technology Center

Applying the SCR Requirements Specification Method to Practical Systems: A Case Study
R. Bharadwaj and C. Heitmeyer, Naval Research Laboratory

574-61
415659
360755
p 26

Risk Knowledge Capture in the Riskit Method

Jyrki Kontio and Victor R. Basili
jyrki.kontio@ntc.nokia.com / basili@cs.umd.edu
University of Maryland
Department of Computer Science
A.V.Williams Building
College Park, MD 20742, U.S.A.
http://www.cs.umd.edu/users/{jkontio, basili}/

Abstract

This paper describes how measurement data and experience can be captured for risk management purposes. The approach presented is a synthesis of the Riskit risk management method and the Experience Factory. In this paper we describe the main goals for risk knowledge capture and derive a classification of information based on those goals. We will describe the Riskit method and its integration with the Experience Factory. We will also outline the initial experiences we have gained from applying the proposed approach in practice.

1. Introduction

Unanticipated problems frequently cause major problems to projects, such as cost overruns, schedule delays, quality problems, and missing functionality. To some degree these problems can be seen as signs of immaturity of our field and we should expect some improvements in our discipline as our methods and knowledge improve. However, as each software development project involves at least some degree of uniqueness and our technology changes continuously, uncertainty about the end results will always accompany software development. While we cannot remove risks from software development, we should learn to manage them better.

Ability to capture, analyze and package experience is a prerequisite for systematic, planned improvements in software engineering [2], as in any field. The framework proposed in this paper builds upon the Riskit method and the Experience Factory, both developed at the University of Maryland. The proposed risk knowledge capture framework contains templates for capturing data about risk elements, templates for capturing relevant information about the risk management process, definition of where in the risk management process risk management knowledge is captured and utilized, and a proposed model for improvement goals for risk management.

2. Background

Risks in software development were not addressed in detail until late 1980's when Boehm [6] proposed and synthesized an approaches for software risk management. His work was complemented by Charette [9], and on these foundations recent advances in software risk management have produced well-documented approaches for risk management [14,18,24,26], several categories of risks have been identified [6,8,23], quantitative approaches for risk management have been proposed and used [5,7,11], and there are several software tools available for risk management. Furthermore, most commonly used software engineering standards [15,16] or assessment frameworks [17,27] require at least some form of risk management to take place.

Despite these efforts and the obvious industry interest in risk management, it seems that few organizations apply specific risks management methods actively [28]. The limited survey data from a recent workshop by Basili and Koji Tori supports this observation: only 20% of respondents claimed to use risk management techniques "extensively" while 40% stated that they are not using "any risk management techniques or approaches" [19]. Clearly, the industrial practice of risks management methods has not yet reached its full potential.

There is little reported work on utilizing data and experience from past project in software engineering risk management literature. Some aspects of Boehm's work implicitly assumed that data from past projects is available if simulation and cost models are used for estimating risks [6]. He also mentioned factors of cost models as possible risk monitoring metrics. Charette has presented an outline of items that should be defined for a project to initiate risk management [10]. He has also given examples of what should be measured and how this data can be graphed for risk management purposes. However, neither one of these approaches can be considered a systematic way to capture or utilize risk management experience.

The Software Engineering Institute (SEI) has collected data from risk assessments they have carried out during the last few years. Their goal seems to be to support analysis risks and their relationships using lexical analysis on the qualitative descriptions in the database [25]. It also seems that frequencies of risks in the database have been used to indicate what are the most common risks. To our knowledge, this database focuses on the results of risk assessments and contains little or no data of what actually happened in projects. Also, it is not clear how much context information is captured about risks and projects so that information in the database can be utilized more effectively.

Hall has defined and implemented a risk database while working at Harris corporation [12]. Risks from three projects were collected [13] and used for analysis in evaluating Hall's risk management maturity model. Hall has also collected survey data on the levels of risks management practices in various organizations [12].

There have been several other, less formal approaches in documenting information about software risks. The ACM SIGSOFT Software Engineering Notes has run a long series of reports on computer related problems or disasters. However, such a list is not very useful for analyzing risks of an individual projects as most of the reported risks do not contain enough context information and details to be useful.

In summary, it seems that while several some advances have been made in the area of software risk knowledge capture, none of the reported approaches provide a comprehensive framework for capturing risk knowledge. Furthermore, software risk management data and knowledge is rarely systematically collected and utilized in the industry. We hope that the framework proposed in this paper can act as a step towards more systematic risk knowledge capture so that our understanding of risks and risk management methods can improve.

3. Risk Knowledge Capture

We have identified three generic types of goals for risk knowledge capture: monitoring risks, understanding risks, and risk management process improvement. First, the risk situation in a project needs to be monitored so that appropriate risk controlling action can be taken. Second, we need to collect information about risks so that frequencies of occurrence and losses of risks can be estimated better. Finally, information needs to be collected so that the risk management process itself can be improved.

Each of the three goals described above focus on different kinds of information and, as always in measurement, the individual metrics and data collection procedures may vary between situations. However, we have identified some generic classes of information based on these three goals. This risk information classification will be introduced in the following paragraphs.

Project context information refers to such information that determines the circumstances and setting where the project is carried out. Project context information is relevant for all software engineering measurement data, but it is particularly important for risk management. The probability of a risk event is often influenced by many factors. By capturing as much as possible of the risk management context information we make it easier to interpret risk management data in the future.

The *risk management infrastructure information* defines what risk management methods, techniques, tools, processes and approaches are used for in risk management. The risk management infrastructure can also be extended to include several other organizational issues that marginally influence risk management, as proposed by Hall [12]. In fact Hall's framework can be used as a model to document the state of risk management infrastructure in an organization.

The *project information* defines the project itself and it includes the definition of the goals, customers, schedule, and constraints of the project. It also includes the definition of the risk management mandate for the project: the risk management mandate is a project-specific statement of the scope of risk management in a project.

	Risk monitoring	Understanding risks	Risk management process improvement
Project context information	X	X	X
Risk management infrastructure information			X
Project information		X	X
Enactment data	X	X	X
Risk management process information			X
Risk element information	X	X	X

Table 1: The relationships between risk knowledge capture goals and risk information types

While the project information provides a static view to the project, *enactment data* provides the dynamic perspective to the project: how much effort is spent, what artifacts are produced and when, how much time has passed, and which individuals worked on the project. Enactment data is usually collected for project control and experience capture purposes as a part of software engineering measurement program.

The *risk management process information* describes the activities and events related to risk management in the project. The risk management process information is, in fact, a special case of project information, but as it represents our special focus, it is meaningful to separate it from the general enactment data of the project.

Finally, *risk element information* refers to information about risks in a project. This type of information can include descriptions of factors that influence risks, such as methods, tools, resources; events that may influence the project; or impacts that risks might have. As we will discuss later, the Riskit method contains conceptual tools to structure such information more formally than is usually done.

The relationships between risk knowledge capture goals and risk information types is presented in Table 1. Each row in Table 1 represents a risk information type and each column a risk knowledge capture goal. An "X" in a cell indicates that the goal in that row normally needs to utilize the type of information listed in that row. However, it is important to point out that information from other categories may often be needed as well, Table 1 merely represents what we believe to be typical relationships between goals and information types.

4. Towards a Risk Knowledge Capture Framework

4.1 The Riskit Method

The Riskit method has been developed to support systematic risk analysis. The Riskit method uses a graphical formalism to support qualitative analysis of risk scenarios before quantification is attempted, its risk ranking approach can be selected based on the availability of history data or accuracy of estimates, it supports multiple goals and stakeholders, and its risk ranking approach is based on the utility theory [20]. We have presented an overview of the activities in the Riskit process in Figure 1. More information about the method is available in separate reports [20-22].

A central part of the Riskit method is the graphical formalism used to document risks, the *Riskit analysis graph.* The Riskit analysis graph is used to define the different aspects of risk explicitly and more formally than is done in casual conversation. The Riskit analysis graph is used during the Riskit process to decompose risks into clearly defined components, *risk elements.* Its components are presented in Figure 2. Each rectangle in the graph represents a risk element and each arrow describes the possible relationship between risk elements. We will define the components of the graph in the following paragraphs.

Instead of informal, general descriptions of risks, we can document the different aspects of risks more precisely, as is shown in Figure 2. The Riskit analysis graph allows explicit and more formal documentation of risks and risk scenarios.

The Riskit method has several potentially useful characteristics that can support risk knowledge capture. First, the Riskit Analysis Graph enforces more formal definition of risks so that more information is collected about each risk. Second, the graphical formalism used as well as the tool that is used to draw these diagrams lay the foundations for automating some of the risk knowledge capture: information about risks can be captured as Riskit graphs are drawn. Third, the Riskit process itself is a defined process that increases repeatability of the risk management process and supports the collection of relevant risk management experience through the templates and guidelines included in the method.

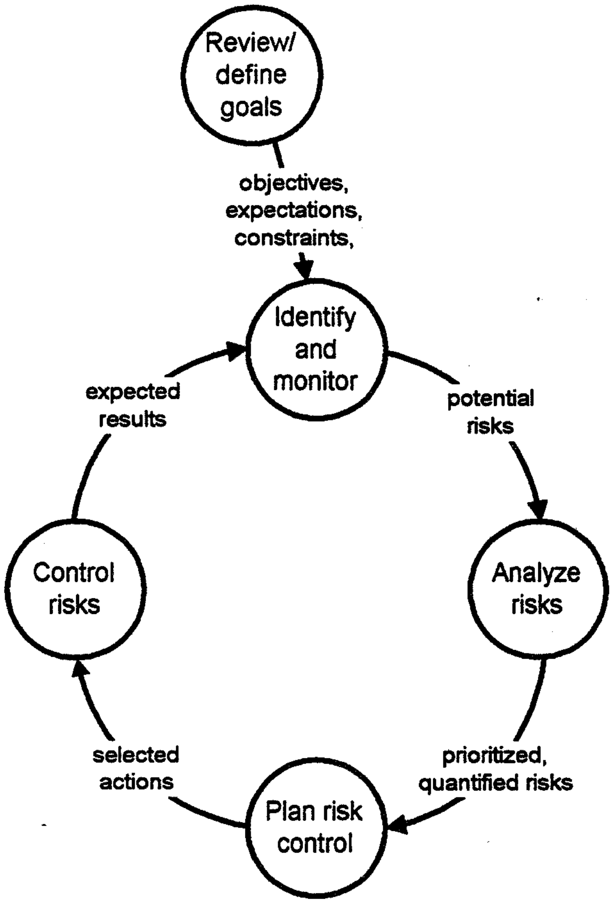


Figure 1: The Riskit risk management cycle[1]

4.2 Risk Knowledge Capture in the Experience Factory Framework

In this section we present how the Riskit method can be integrated into Basili's Experience Factory (EF) and Quality Improvement Paradigm (QIP) [3,4]. The Quality Improvement Paradigm (QIP) is a systematic process for continuous improvement. It is similar to the scientific principle of learning in its emphasis of learning through empirical experience. The QIP process can be seen as consisting of three main

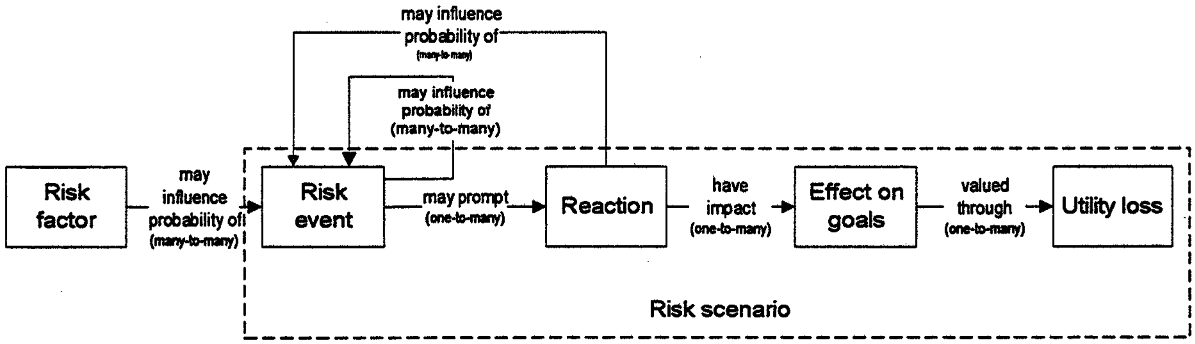

may influence probability of (many-to-many)
may influence probability of (many-to-many)
Risk factor
may influence probability of (many-to-many)
Risk event
may prompt (one-to-many)
Reaction
have impact (one-to-many)
Effect on goals
valued through (one-to-many)
Utility loss
Risk scenario

Figure 2: A conceptual view of the elements in the Riskit analysis graph

[1] Note that Figure 1 presents a simplified view of the activities in the Riskit process. More comprehensive description of the Riskit process is available through other publications [20].

activities that include the six steps normally described for QIP: *planning*, consisting of the steps characterize, set goals, and choose process; *execute*; and *learning*, consisting of steps analyze and package [4].

The Experience Factory Organization is an organizational model for implementing the QIP process. The main idea of this approach is the recognition the distinct roles belonging to the project organization and a learning organization, the Experience Factory. The Project Organization focuses on delivering the software product and the Experience Factory focuses on learning from experience and improving software development practice in the organization. A central aspect of the Experience Factory is the Experience Base, a repository of data and knowledge about the software development process and products. The knowledge in the Experience Base can be in various forms, it can include raw and summarized data, mathematical models about the data (e.g., prediction models), experiment reports, and qualitative lessons learned reports [1-4].

From risk management perspective the Experience Factory concept serves to fulfill the following goals:

- separation of responsibilities between risk management within projects and improving the risk management process itself and improving the understanding of risks;
- systematic capture and accumulation of risk management knowledge into the Experience Base;
- continuous learning from risk management experience through measurement, data collection, analysis and synthesis; and
- systematic reuse of accumulated risk management knowledge through packaging and dissemination of this knowledge.

When the Riskit process is viewed from the perspective of the Experience Factory and the QIP cycle, it is possible to identify steps where risk management process needs to be initiated to support the QIP process, as shown in Figure 3. The initial planning cycle represents the first cycle of the Riskit process, whereas the risk management cycle supporting the execute step support mainly project monitoring, i.e., risk monitoring and control. The learning step analyzes and packages the risk management experience gained through the process.

All of the QIP and Riskit activities represented in Figure 3 produce data about risk management that can be captured and stored in an experience base. We have defined a database definition for such information for the Riskit process. Furthermore, the project planning step in QIP also includes goal definition for risk understanding and risk management process improvement. These goals can introduce new data and experience capture needs that can be implemented as required. The learning step of QIP, and the two risk related activities associated with it, utilize the data and experience collected about risks and produce packaged, reusable pieces of risk knowledge to be stored in the Experience Base and utilized in future projects.

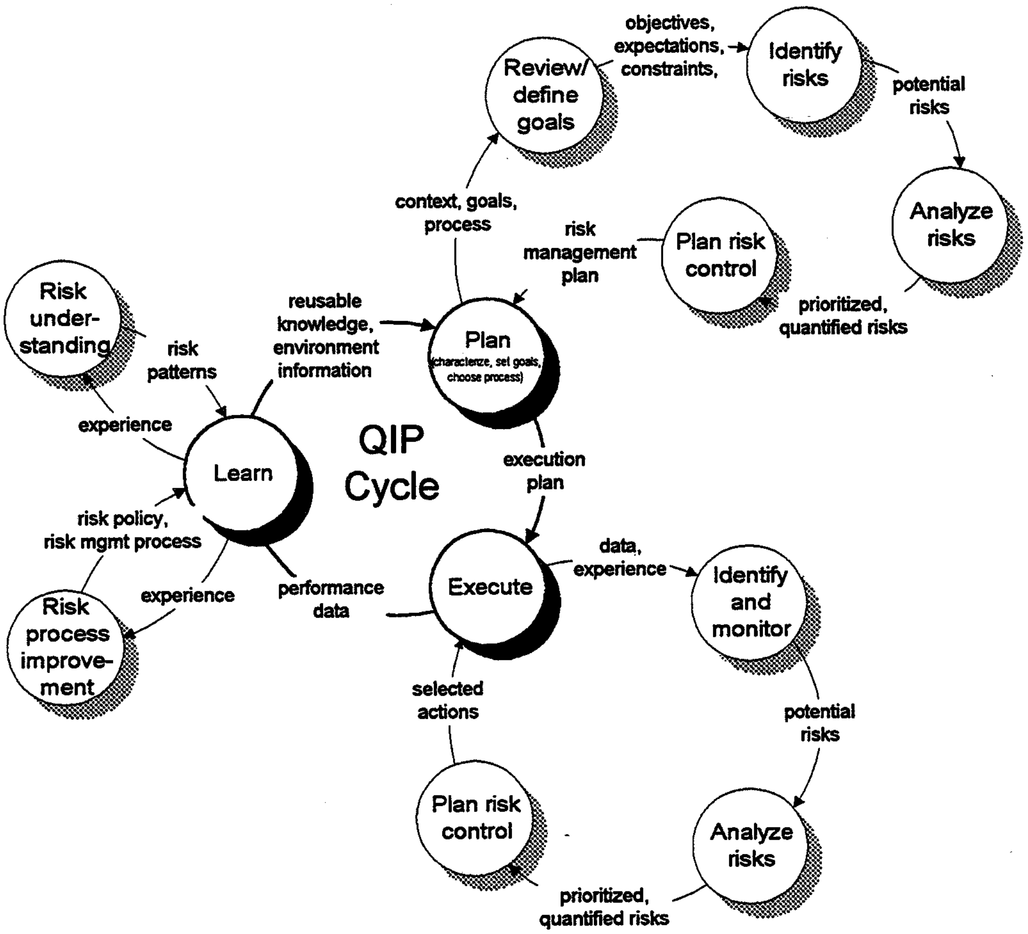


Figure 3: The mapping between QIP cycle and the Riskit process

4.3 Applying the Riskit Knowledge Capture Framework

The Riskit method and its knowledge capture framework have been applied in several trial projects. So far the case studies have focused on the last one of the goals we introduced earlier: improving the method itself.

The goals of the first case study [22] were to characterize the method, investigate its feasibility, and to collect empirical feedback on its use to be able to improve it. This first case study resulted in several changes in the method itself and it produced approximately 15 risk scenarios (corresponding to about 50 risk elements). Project and context information was documented informally in a separate report [22]. Other, on-going empirical studies with the method focus similarly on obtaining feedback on the methods feasibility and effectiveness.

These case studies have produced large amounts of risk management data and experience and we are in the process of formalizing this data into a risk management database, or a risk management experience base. Our goal is to evaluate the feasibility and potential benefits of such a database given the empirical data we have obtained.

5. Conclusions

This paper presented background and motivation for risk knowledge capture and proposed a classification of goals and information types for such capture. We also outlined how the Riskit method supports this type of experience capture. We reported some initial experiences from the use of the Riskit method and the proposed risk knowledge capture framework.

The potential benefits from risk knowledge capture are significant. Frequency and severity of typical risks can be estimated more accurately, changes in potential risks observed more concretely, risk management methods and tools can be improved based on empirical feedback, and projects have more up-to-date information about risks and risk management actions in a project. Furthermore, it may be possible to identify and package some risk management patterns: reusable pieces of risk management knowledge that can be utilized by project managers. Examples of such risk patterns could be lists of risks that are associated with certain project characteristics and descriptions of risk controlling actions that have been found effective in controlling certain types of risks. The Riskit method itself, through its more formal definition of risk and its graphical representation formalism, provides a good basis to capture and reuse such knowledge in practice.

While it is too early to make any conclusions about the feasibility and benefits of the proposed risk knowledge capture approach, the combination of Riskit and the Experience Factory contain the necessary foundations for more systematic and detailed experience capture. The initial empirical studies indicate that the approach is feasible in industrial context.

However, it is yet to be determined whether such experience capture is cost effective. Although the Riskit method may potentially allow automation of some of the experience capture processes, it is currently a manually driven process and therefore potentially too costly in large scale use. Furthermore, given the subjective nature of the definition of risk, one could also question how reliable is experience that, to a large degree, is based on subjective opinions and judgment calls about future events.

While there may be some valid concerns about the cost-effectiveness of a risk management database and its utilization, it is nevertheless likely that risk management experience needs to be captured and formulated into knowledge to be reused in future projects. The Riskit method provides a more concrete basis even for qualitative knowledge formulation process, even when the risk management experience and data are not captured into a formal database but stored in less formal parts of the Experience Base.

6. References

[1] V. R. Basili, Quantitative Evaluation of Software Engineering Methodology, 1985. Proceedings of the First Pan Pacific Computer Conference. Also available as computer science technical report TR-1519, University of Maryland.

[2] V. R. Basili, Software Development: A Paradigm for the Future, 1989. Proceedings of the 13th Annual Computer Software and Applications Conference (COMPSAC).

[3] V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora, The Software Engineering Laboratory - an Operational Software Experience Factory, pp. 370-381, 1992. Proceedings of the International Conference on Software Engineering, May 1992.

[4] V. R. Basili, G. Caldiera, and H. D. Rombach. The Experience Factory. In: *Encyclopedia of Software Engineering*, Anonymous New York: John Wiley & Sons, 1994.pp. 470-476.

[5] J. Berny and P. R. F. Townsend, Macrosimulation of project risks -- a practical way forward, *International Journal of Project Management*, vol. 11, pp. 201-208, 1993.

[6] B. W. Boehm. *Tutorial: Software Risk Management*, IEEE Computer Society Press, 1989. pp. 1-469.

[7] J. A. Bowers, Data for project risk analyses, *International Journal of Project Management*, vol. 12, pp. 9-16, 1994.

[8] M. J. Carr, S. L. Konda, I. A. Monarch, F. C. Ulrich, and C. F. Walker. *Taxonomy-Based Risk Identification, SEI Technical Report SEI-93-TR-006*, Pittsburgh, PA: Software Engineering Institute, 1993.

[9] R. N. Charette. *Software Engineering Risk Analysis and Management*, New York: McGraw-Hill, 1989.

[10] R. N. Charette. *Applications Strategies for Risk Analysis*, New York: McGraw-Hill, 1990.

[11] R. Fairley, Risk Management for Software Projects, *IEEE Software*, vol. 11, pp. 57-67, 1994.

[12] E. M. Hall, Proactive Risk Management Methods for Software Engineering Excellence 1995. Florida Institute of Technology. Also available from UMI Dissertation Services.

[13] E. M. Hall, Email correspondence ed. J. Kontio. 1996. email correspondence.

[14] R. Hefner, Experience with Applying SEI's Risk Taxonomy, 1994. Proceedings of the Third SEI Conference on Software Risk Management. SEI. Pittsburgh, PA.

[15] IEEE. *IEEE Standard for Developing Software Life Cycle Processes*, New York: IEEE Computer Society, 1992.

[16] ISO. *ISO 9000-3, Guidelines for the application of ISO 9001 to the development, supply and maintenance of software, ISO 9000-3:1991(E)*, International Standards Organization, 1991.

[17] ISO. *SPICE: Baseline Practices Guide, an unfinished draft of a standard being developed for ISO, version 1.00*, 1994. (UnPub)

[18] D. W. Karolak. *Software Engineering Risk Management*, Washington, DC: IEEE, 1996.

[19] J. Kontio, IWSED-95 Web pages Anonymous. Anonymous. *<None Specified>*, vol. 1995. University of Maryland. World Wide Web. http://www.cs.umd.edu/projects/SoftEng/ESEG/iwsed/iwsed95/.

[20] J. Kontio, The Riskit Method for Software Risk Management, version 1.00 Anonymous1996. Computer Science Technical Reports. University of Maryland. College park, MD.

[21] J. Kontio and V. R. Basili, Empirical Evaluation of a Risk Management Method, 1997. Proceedings of the SEI Conference on Risk Management. Software Engineering Institute. Pittsburgh, PA.

[22] J. Kontio, H. Englund, and V. R. Basili, Experiences from an Exploratory Case Study with a Software Risk Management Method Anonymous CS-TR-3705, 1996. Computer Science Technical Reports. University of Maryland. College Park, Maryland.

[23] L. Laitinen, S. Kalliomäki, and K. Känsälä. *Ohjelmistoprojektien Riskitekijät, Tutkimusselostus N:o L-4*, Helsinki: VTT, Tietojenkäsittelytekniikan Laboratorio, 1993.

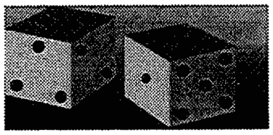
[24] J. V. Michaels. *Technical Risk Management*, Upper Saddle River, NJ: Prentice Hall, 1996.

[25] I. A. Monarch, S. L. Konda, and M. J. Carr, Software Engineering Risk Repository, 1996. Proceedings of the 1996 SEPG Conference. Software Engineering Institute. PIttsburgh, PA.

[26] G. Pandelios, T. P. Rumsey, and A. J. Dorofee, Using Risk Management for Software Process Improvement, 1996. Proceedings of the 1996 SEPG Conference. SEI. Pittsburgh.

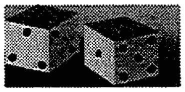
[27] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. *Capability Maturity Model for Software, Version 1.1, Technical Report SEI-93-TR-024*, Pittsburgh: Software Engineering Institute, Carnegie Mellon University, 1993.

[28] J. Ropponen, Risk Management in Information System Development Anonymous TR-3, 1993. Computer Science Reports. University of Jyväskylä, Department of Computer Science and Information Systems. Jyväskylä.



Risk Knowledge Capture in the Riskit Method

Jyrki Kontio and Victor R. Basili
University of Maryland
Department of Computer Science
A.V.Williams Building
College Park, MD 20742, U.S.A.
{jkontio, basili}@cs.umd.edu
http://www.cs.umd.edu/users/jkontio/

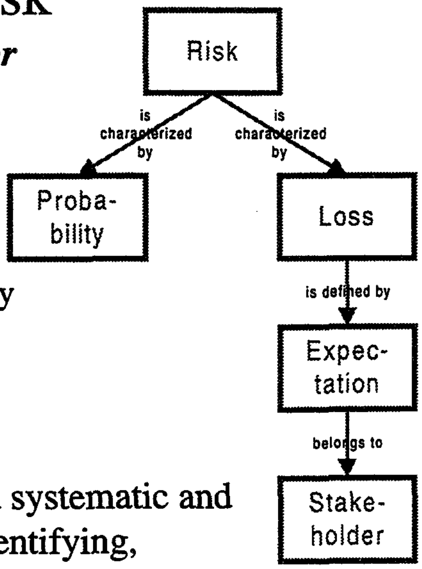


Outline

- Definition of risk
- The Riskit method
  - ◆Underlying principles
  - ◆Riskit process through an example
- Case studies
- Conclusions

Definitions of Risk

- **Risk:** ***a possibility of loss -- or any characteristic, object or action that is associated with that possibility.***
- Risk is associated with:
  - **probability**: there is uncertainty
  - **loss**: some harm or damage
    - goals or expectations
    - stakeholder



Risk

is characterized by

is characterized by

Proba-bility

Loss

is defined by

Expec-tation

belongs to

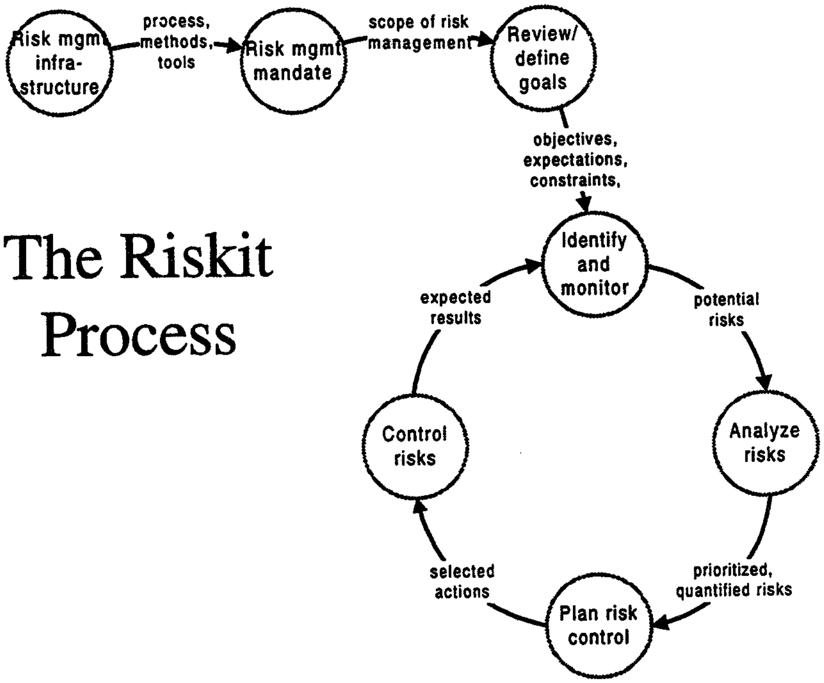
Stake-holder

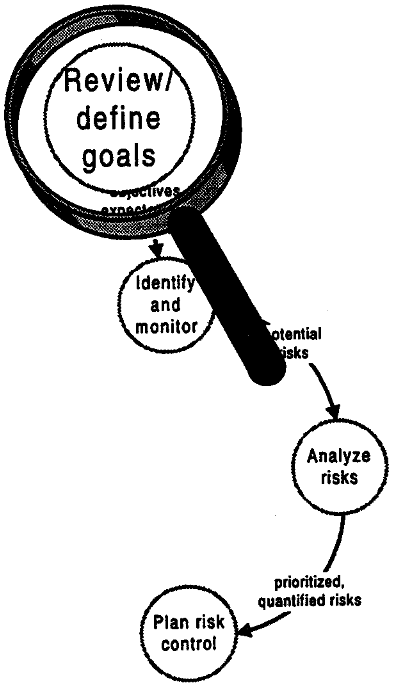
- ***Risk management*** refers to a systematic and explicit approach used for identifying, analyzing and controlling risk.

Riskit Main Principles

- Risks are relative to goals and expectations
- There's always more than one stakeholder
- Risks must be well defined
- Multiple goal effects are accounted for
- Losses estimated through utility loss
- Learn from past experience

The Riskit Process





Example

- This presentation
- Stakeholders
  - Audience
  - Presenter
  - Session chair

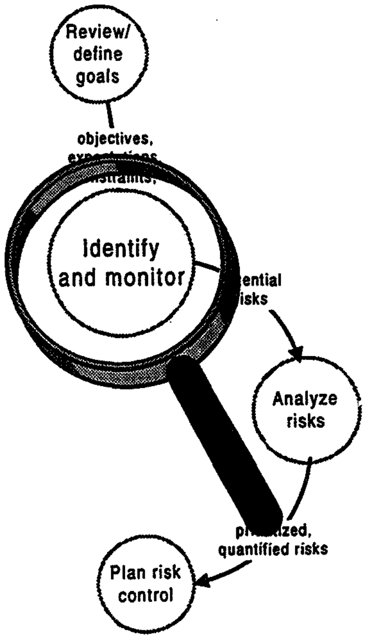
"There's always more than one stakeholder"

- Goals
  - Learn about risk management
  - Finish in 30 minutes
  - Sell Riskit to practitioners

"Risks are relative to goals and expectations"

Example: Review and Definition of Goals

Goal	Stakeholders	Metrics	Target
Learn about risk mgmt	• Audience	• Feedback • Questions asked • Use of Riskit?	
Finish in 30 mins	• Audience • Session chair	• Elapsed time	30 minutes
"Sell" Riskit	• Presenter	• Feedback • Questions asked • Info requests • WWW visits...	Some will try it out

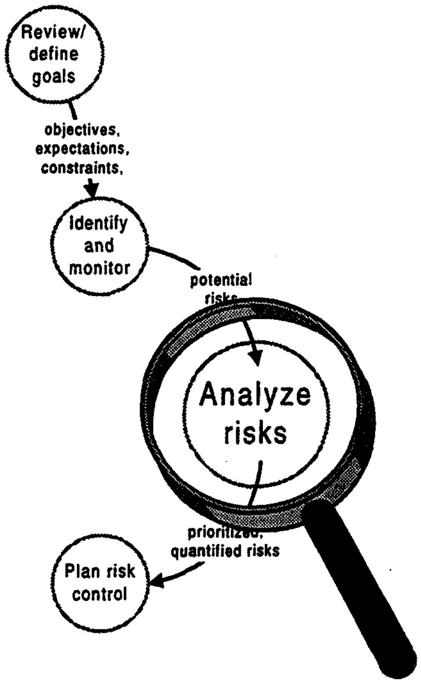


Example: Risk Identification

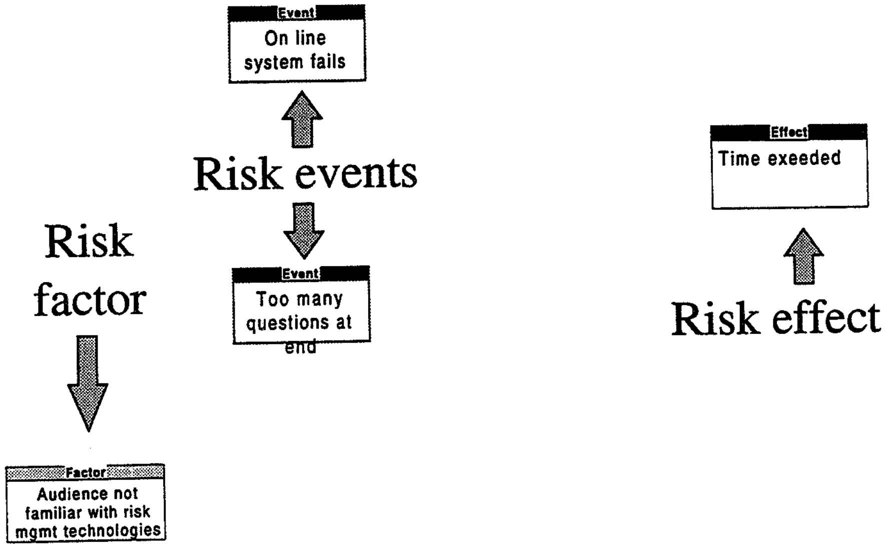
- Possible risks:
  - Talk will last longer than 30 minutes
  - On line slide presentation system fails
  - Presenter will mess up his slides
  - Too many questions at the end
  - Presenter will ramble off the topic
  - Audience does not have much background in risk management
  - Booster rockets from the space shuttle hit this building

Example: Risk Identification

- Selected risks for risk analysis:
  - **Talk will last longer than 30 minutes**
  - **On line slide presentation system fails**
  - Presenter will mess up his slides
  - **Too many questions at the end**
  - Presenter will ramble off the topic
  - **Audience does not have much background in risk management**
  - Booster rockets from the space shuttle hit this building
  - ...



Example: Risk Scenarios



Event
On line system fails

Risk events

Event
Too many questions at end

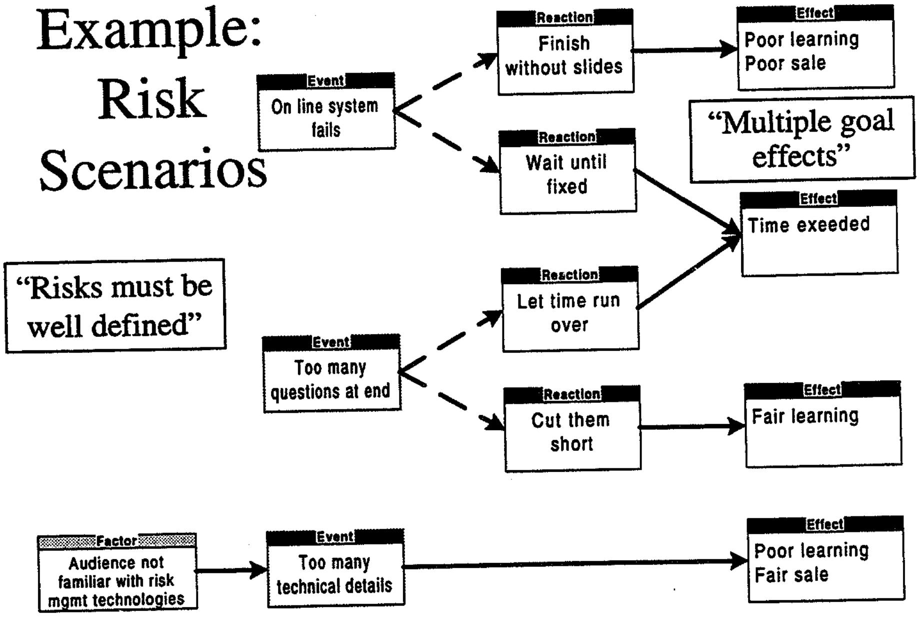
Risk factor

Factor
Audience not familiar with risk mgmt technologies

Effect
Time exeeded

Risk effect

Example: Risk Scenarios



Event
On line system fails

Reaction
Finish without slides

Effect
Poor learning
Poor sale

Reaction
Wait until fixed

"Multiple goal effects"

Effect
Time exeeded

"Risks must be well defined"

Reaction
Let time run over

Event
Too many questions at end

Reaction
Cut them short

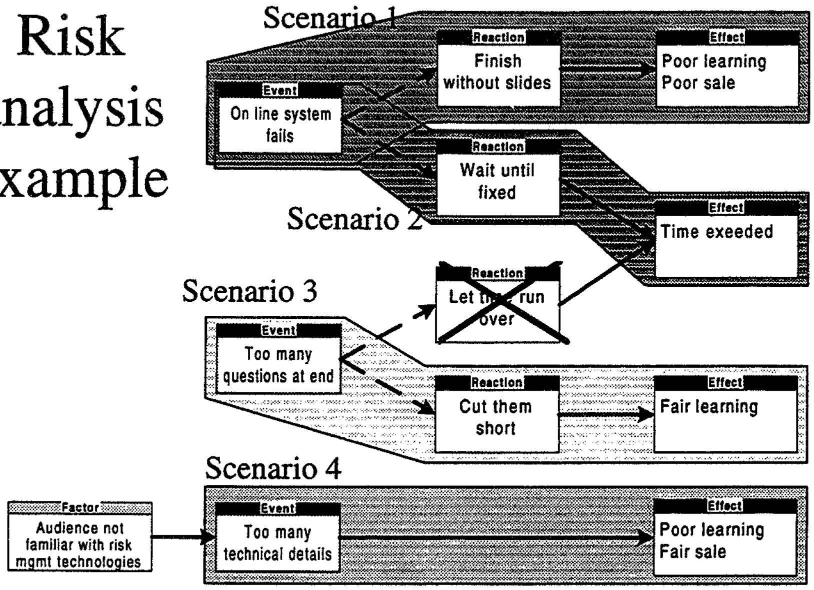
Effect
Fair learning

Factor
Audience not familiar with risk mgmt technologies

Event
Too many technical details

Effect
Poor learning
Fair sale

Risk analysis example



Scenario 1

Scenario 2

Scenario 3

Scenario 4

Event
On line system fails

Reaction
Finish without slides

Effect
Poor learning
Poor sale

Reaction
Wait until fixed

Effect
Time exeeded

Event
Too many questions at end

~~Reaction~~
~~Let time run over~~

Reaction
Cut them short

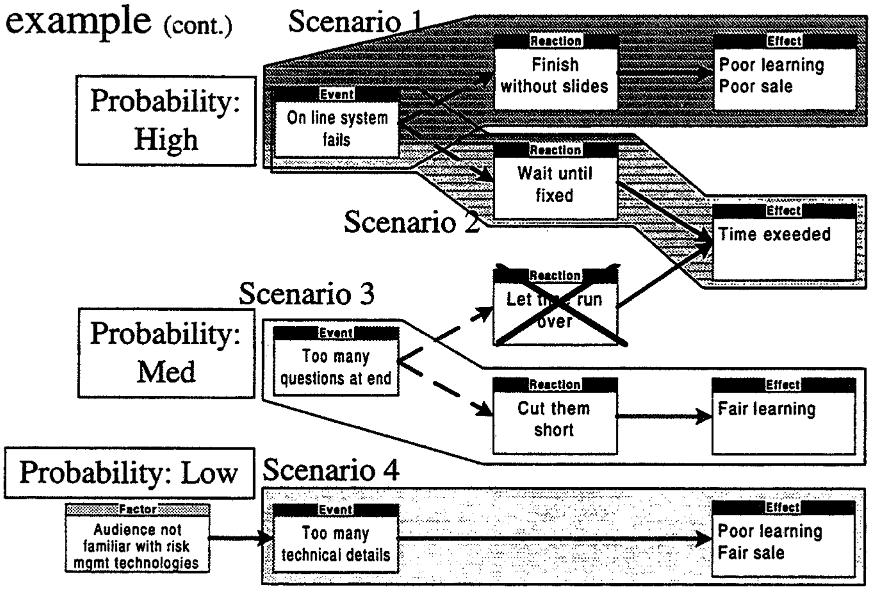
Effect
Fair learning

Factor
Audience not familiar with risk mgmt technologies

Event
Too many technical details

Effect
Poor learning
Fair sale

Risk analysis example (cont.)



Scenario 1

Scenario 2

Scenario 3

Scenario 4

Probability: High

Probability: Med

Probability: Low

Event
On line system fails

Reaction
Finish without slides

Effect
Poor learning
Poor sale

Reaction
Wait until fixed

Effect
Time exeeded

Event
Too many questions at end

~~Reaction~~
~~Let time run over~~

Reaction
Cut them short

Effect
Fair learning

Factor
Audience not familiar with risk mgmt technologies

Event
Too many technical details

Effect
Poor learning
Fair sale

Ranking Risk Effects

"Losses estimated through utility loss"

Stakeholders: Effects:	Audience	Presenter	Session Chair
Poor learning Poor sale	■	■	Low
Time exceeded	Med	Low	■
Fair learning	Med	■	Low
Poor learning Fair sale	■	Med	Low

Example:

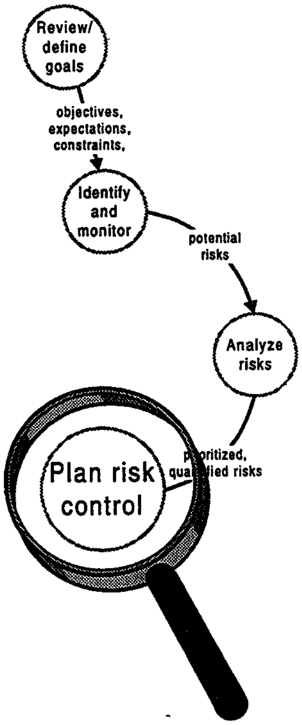
Selecting the scenarios

Presenter	Loss High	Loss Med	Loss Low
Prob High	Scenario 1		Scenario 2
Prob Med	Scenario 3		
Prob Low		Scenario 4	

Audience	Loss High	Loss Med	Loss Low
Prob High	Scenario 1	Scenario 2	
Prob Med		Scenario 3	
Prob Low	Scenario 4		

Chair	Loss High	Loss Med	Loss Low
Prob High	Scenario 2		Scenario 1
Prob Med			Scenario 3
Prob Low			Scenario 4

I	II	III
II	III	IV
III	IV	V



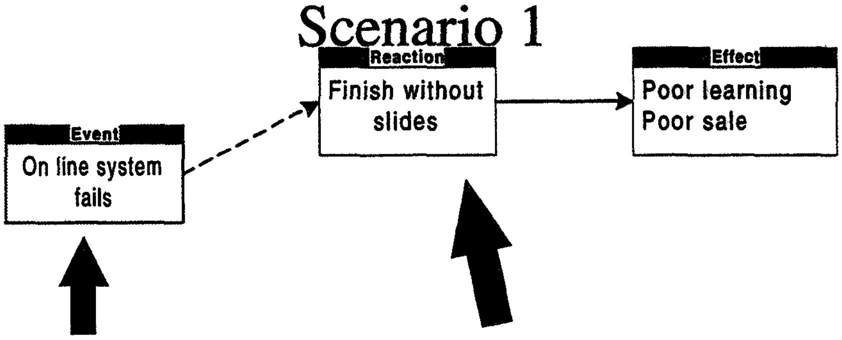
Risk Control Planning

- Presenter's priorities:
  - Scenario 1
  - Scenario 3
  - Scenario 2
  - Scenario 4
- Audience's priorities:
  - Scenario 1
  - Scenario 2
  - Scenario 3 and 4
- Chair's priorities
  - Scenario 2
  - Scenario 1
  - Scenario 3
  - Scenario 4

04-Dec-96

- ■ Joint risk control for Scenario 1 and Scenario 2
- ■ Scenario 3 is presenter's problem (and so is scenario 4)

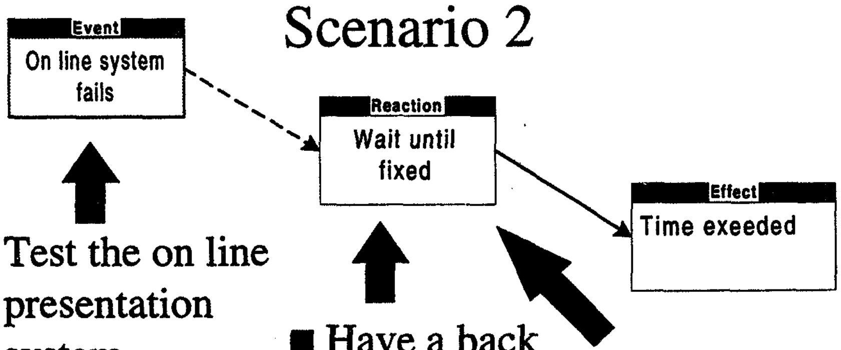
Risk Control Planning for Scenario 1



- Test the on line presentation system thoroughly

■ Bring back up slides for overhead

Risk Control Planning for Scenario 2

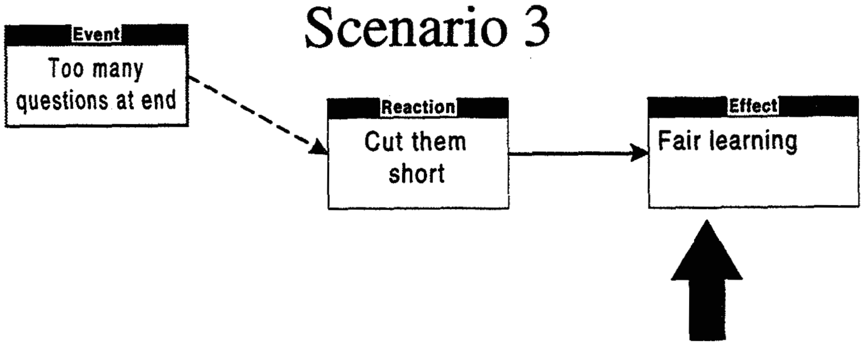


- Test the on line presentation system thoroughly

■ Have a back up system ready

■ Bring back up slides for overhead

Risk Control Planning for Scenario 3

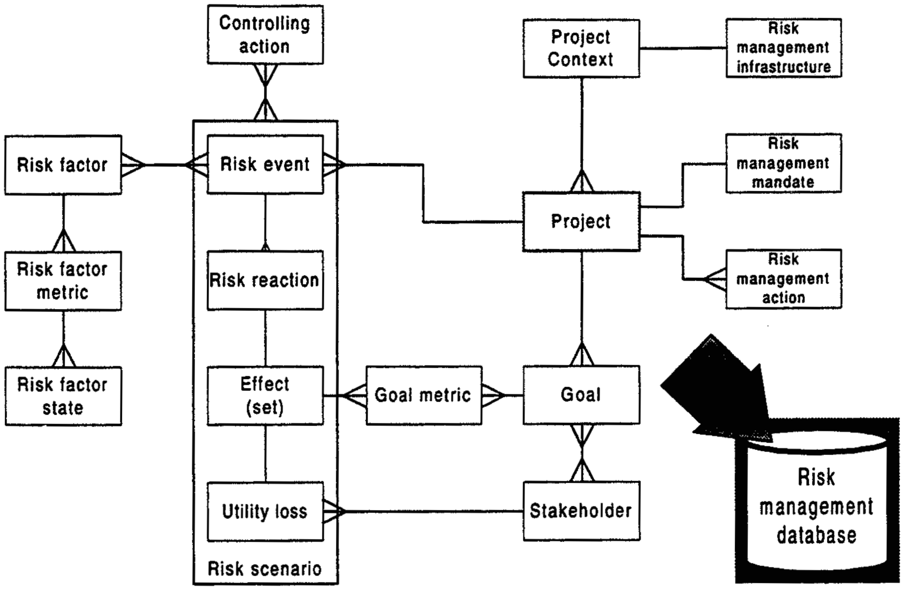


- Provide references for further information
- Hang around after the talk

Risk Management Experience Capture

- Goals
  - Risk management process improvement
  - Risk understanding
  - Risk monitoring
- Means
  - Risk management Experience Base
  - Risk management experience analysis

Risk Management Experience Base



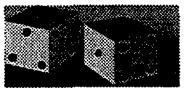
Empirical Studies

- SEL Case Study
  - exploratory study to support method development
- Hughes Case Study
  - exploratory study on method use
  - describe the method, assess feasibility, compare effectiveness
  - Produced 4 stakeholders, 17 goals and 48 risks

Case Study Experiences

- Riskit results in more detailed description and analysis of risks
- Method users gave high marks for Riskit for
  - "Well-defined process, usable and practical"
  - "Provides a high-level view of all risks"
  - "More confidence in results, more thorough, more complete analysis"
- Identified risks that normal approach might have ignored
- Riskit consumed more resources

Conclusions



- Benefits
  - avoids common limitations in risk management (multiple goals and stakeholders, risk ranking)
  - explicit and precise description of risks
  - increases user confidence in results
  - captures risk management experience
- Potential problems
  - higher cost
- Further work
  - case studies continue (e.g. Nokia Corporation)
  - potential automation for graphs and database

Main References

V. R. Basili, Software Development: *A Paradigm for the Future* (keynote address), 1989. Proceedings of the 13th Annual Computer Software and Applications Conference (COMPSAC).

B. W. Boehm. *Tutorial: Software Risk Management*, B.W. Boehm (Ed). IEEE Computer Society Press, 1989.

R. N. Charette. *Software Engineering Risk Analysis and Management*, New York: McGraw-Hill, 1989.

J. Kontio, *The Riskit Method for Software Risk Management, version 1.00* 1996. Computer Science Technical Reports. University of Maryland. College park, MD.

J. Kontio and H. Englund, *Experiences from an Exploratory Case Study with a Software Risk Management Method* 1996. Computer Science Technical Reports. University of Maryland. College Park, Maryland.

More information:

- email: jkontio@cs.umd.edu
- WWW: http://www.cs.umd.edu/users/jkontio/

S15-61
415660
360759
p18

Requirement Metrics for Risk Identification

Theodore Hammer, GSFC, 301-614-5225, Theodore.Hammer@gsfc.nasa.gov
Lenore Huffman, SATC, 301-286-0099, Lenore.Huffman@gsfc.nasa.gov
William Wilson, SATC, 301-286-0102, William.Wilson@gsfc.nasa.gov
Dr. Linda Rosenberg, SATC, 301-286-0087, Linda.Rosenberg@gsfc.nasa.gov
Lawrence Hyatt, GSFC, 301-286-7475, Larry.Hyatt@gsfc.nasa.gov

1. Introduction

The Software Assurance Technology Center (SATC) is part of the Office of Mission Assurance of the Goddard Space Flight Center (GSFC). The SATC's mission is to assist National Aeronautics and Space Administration (NASA) projects to improve the quality of software which they acquire or develop. The SATC's efforts are currently focused on the development and use of metric methodologies and tools that identify and assess risks associated with software performance and scheduled delivery. This starts at the requirements phase, where the SATC, in conjunction with software projects at GSFC and other NASA centers is working to identify tools and metric methodologies to assist project managers in identifying and mitigating risks. This paper discusses requirement metrics currently being used at NASA in a collaborative effort between the SATC and the Quality Assurance Office at GSFC to utilize the information available through the application of requirements management tools.

Requirements development and management have always been critical in the implementation of software systems - engineers are unable to build what analysts can not define. Recently, automated tools have become available to support requirements management. The use of these tools not only provides support in the definition and tracing of requirements, but also opens the door to effective use of metrics in characterizing and assessing risks. Metrics are important because of the benefits associated with early detection and correction of problems with requirements; problems not found until testing are at least 14 times more costly to fix than problems found in the requirements phase. This paper discusses two facets of the SATC's efforts to identify requirement risks early in the life cycle, thus preventing costly errors and time delays later in the life cycle.

The first effort that will be discussed is the development and application of an early life cycle tool for assessing requirements that are specified in natural language. This paper describes the development and experimental use of the Automated Requirements Measurement (ARM) tool. Reports produced by the tool are used to identify specification statements and structural areas of the requirements document which need to be improved.

The second effort discusses metrics analysis of information in the requirements database used to provide insight into the stability and expansion of requirements. The research into attaching certain document attributes to analyses results done on requirements stored in requirements databases is providing project management with valuable information. The correlations between document structure and language, and requirement expansion and testing have been strong. This information has been assisting and continues to assist the Quality Assurance Office in its project oversight role.

When discussing metric results the project must remain anonymous; however, for this paper, a general understanding of the project's development environment is necessary. The project in discussion is implementing a large system in three main incremental builds.[1] The development of these builds is overlapping, e.g. design and coding of the second and third builds started prior to the completion of the first build. Each build adds new functionality to the previous build and satisfies a further set of requirements. The definition of requirements for this system started with the formulation of System Level Requirements. These are mission level requirements for the space craft and ground system; they are at a very high level and rarely, if ever, change. Requirements at this level will not be discussed since they are not stored in the requirements database under scrutiny.

System requirements then undergo several levels of decomposition to produce Top Level Requirements. These requirements are also high level and change should be minimal. The development of the project discussed in this paper started with the Top Level requirements. Top Level requirements are then divided into subsystems and a further level is derived in greater detail; hence, "Specification Requirements". Generally, contracts are bid using this level of requirement detail. The Design Requirements are derived from the Specification requirements; these requirements are the ones used to design and code the system. This project chose to develop an additional intermediate set of Specification Level Requirements after contract award.

2. Automated Requirements Measurement Tool (ARM)

Despite the significant advantages attributed to the use of formal specification languages, their use has not become common practice. Because requirements that the acquirer expects the developer to contractually satisfy must be understood by both parties, specifications are most often written in natural language. The use of natural language to prescribe complex, dynamic systems has at least three severe problems: ambiguity, inaccuracy and inconsistency. Many words and phrases have dual meanings which can be altered by the context in which they are used. Weak sentence structure can also produce ambiguous statements. For example, the statement "Twenty seconds prior to engine shutdown anomalies shall be ignored." could result in at least three different implementations. Defining a large, multi-dimensional capability within the limitations imposed by the two dimensional structure of a document can obscure the relationships between individual groups of requirements.

The SATC developed the Automated Requirements Measurement (ARM) tool to address certain management needs: that of providing metrics which NASA project managers can use to assess the quality of their requirements specification documents and that of identifying risks poorly specified requirements introduce into any project. The ARM tool searches the requirements document for terms the SATC has identified as quality indicators. Reports produced by the tool are used to identify specification statements and structural areas of the

[1] Various names are used, deliveries, releases, builds, but the term build will be used in this paper.

requirements document which need improvement. It must be emphasized that the tool does not assess correctness of the requirements specified; it does, however, assess the structure, language, and vocabulary of both the document itself and the individual requirements.

2.1 Specification Quality Attributes

The SATC study was initiated by compiling the following list of quality attributes that requirements specifications are expected to exhibit: Completeness, Consistency, Correctness, Modifiability, Ranking, Traceability, Non-ambiguity, and Verifiability. As a practical matter, it is generally accepted that requirements specifications should also be Valid and Testable. These characteristics are not independent. A specification, obviously, cannot be correct if it is incomplete or inconsistent.

Most, if not all, of these quality attributes are subjective. A conclusive assessment of a requirements specification's appropriateness requires review and analysis by technical and operational experts in the domain addressed by the requirements. Several of these quality attributes, however, can be linked to primitive indicators that provide some evidence that the desired attributes are present or absent.

2.2 Specification Quality Indicators

Although most of the quality attributes of documented requirements are subjective, there are aspects of the documentation which can be measured and therefore can be used as indicators of quality attributes. Nine categories of quality indicators for requirement documents and specification statements were established for two types of classification: those related to the examination of individual specification statements, and those related to the requirements document as a whole. The categories related to individual specification statements are: Imperatives, Continuances, Weak Phrases, Directives, and Options. The categories of indicators related to the entire requirements document are: Size, Specification Depth, Readability, and Text Structure.

- IMPERATIVES are those words and phrases that command that something must be provided. "Shall" normally dictates the provision of a functional capability; "Must" or "must not" normally establishes performance requirements or constraints; "Will" normally indicates that something will be provided from outside the capability being specified. The ARM report lists the imperatives and their associated counts in descending order of forcefulness. An explicit specification will have most of its counts high in the report IMPERATIVE list (i.e. shall, must, required).
- CONTINUANCES are phrases such as "the following:" that follow an imperative and precede the definition of lower level requirement specification. The extent that CONTINUANCES are used is an indication that requirements have been organized and structured. These characteristics contribute to the tractability and maintenance of the subject requirement specification. However, extensive use of continuances indicate multiple, complex requirements that may not be adequately factored into development resource and schedule estimates.

- WEAK PHRASES are clauses that are apt to cause uncertainty and leave room for multiple interpretations. Use of phrases such as "adequate" and "as appropriate" indicate that what is required is either defined elsewhere or worst, the requirement is open to subjective interpretation. Phrases such as "but not limited to" and "as a minimum" provide the basis for expanding requirements that have been identified or adding future requirements. WEAK PHRASE total is indication of the extent that the specification is ambiguous and incomplete.
- DIRECTIVES are words or phrases that indicate that the document contains examples or other illustrative information. DIRECTIVES point to information that makes the specified requirements more understandable. The implication is the higher the number of Total DIRECTIVES the more precisely the requirements are defined.
- OPTIONS are those words that give the developer latitude in the implementation of the specification that contains them. This type of statement loosens the specification, reduces the acquirer's control over the final product, and establishes a basis for possible cost and schedule risks.
- LINES OF TEXT are the number of individual lines of text read by the ARM program from the source file.
- UNIQUE SUBJECTS is the count of unique combinations and permutations of words immediately preceding imperatives in the source file. This count is an indication of the scope of the document. The ratio of unique subjects to the total for SPECIFICATION STRUCTURE is also an indicator of the specifications' detail.
- READABILITY STATISTICS are a category of indicators that measure how easily an adult can read and understand the requirements document. Flesch-Kincaid Grade Level index is also based on the average number of syllables per word and the average number of words per sentence. (For the project of this paper, the score indicates a grade school level.)

Table 1 below shows the summary statistics for 41 NASA requirement documents and the results for the project discussed in this paper, Project X.

41 DOCUMENTS	**Lines of Text** - Count of the physical lines of text	**Imperatives** - shall, must, will, should, is required to, are applicable, responsible for	**Continuances** - as follows, following, listed, Inparticular, support	**Weak Phrases** - adequate, as applicable, as appropriate, as a minimum, be able to, be capable, easy, effective, not limited to, if practical	**Directives** - figure, table, for example, note:	**Options** - can, may, optionally	**Subjects** - Count fo unique identifiers preceding imperatives	**Flesch-Kincaid Grade Lvl** - Readability Index
Median	927.0	192.5	70.5	24.0	13.0	20.0	76.0	11.4
Mean	1,569.9	415.1	155.4	41.0	25.0	39.6	173.5	10.8
Max	7,499.0	2,004.0	1,023.0	249.0	119.0	169.0	804.0	13.8
Min	36.0	25.0	8.0	0.0	0.0	0.0	12.0	7.8
Stdev	1,758.3	468.7	202.2	51.3	28.7	45.4	203.2	1.6
Project X	11,596	1,982	620	374	132	177	510	9

Table 1: Summary Statistics

Two approaches can be applied to compare Project X to the metric database containing 41 documents. The first approach is to compare Project X to the other projects using standard deviations. Since approximately 99% of the projects should fall within +/- 3 standard deviations, we mark that range on the graph in Figure 1.

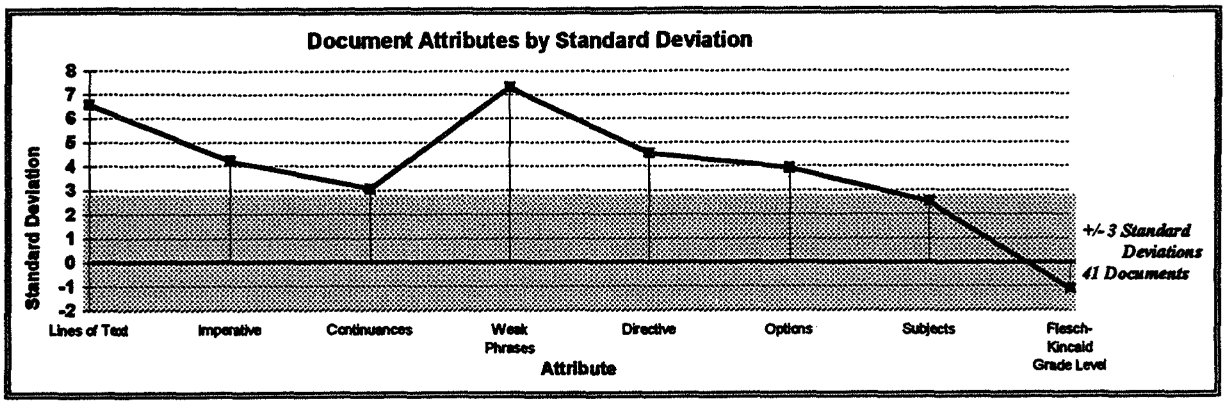


Figure 1: Document Attributes by Standard Deviation

However, since Project X is larger than all projects analyzed to date, Figure 1 may present an inaccurate picture. Normalizing the data on Lines of Text (Figure 2) yields a different picture of Project X in relation to other projects, thus suggesting that Project X attribute counts are in line. The number of weak phrases should however be investigated since it indicates potential risk.

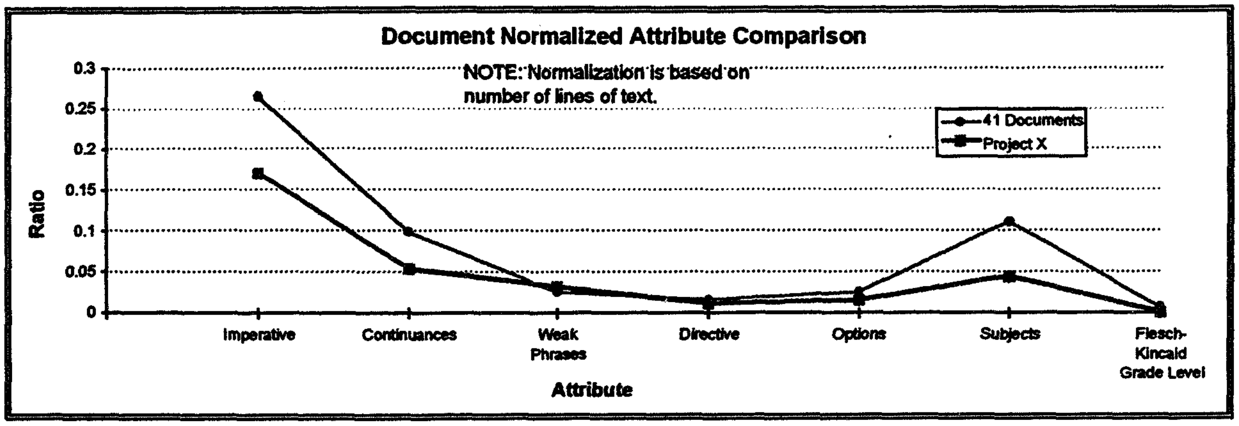


Figure 2: Document Attributes Normalized by Lines of Text

The structure of the document is also indicative of potential project risks. ARM uses the structure depth and specification depth to depict two aspects of the document's structure.

- STRUCTURE DEPTH provides a count of the numbered statements at each level of the source document. These counts provide an indication of the document's organization and consistency and level of detail. High level specifications will usually not have numbered statements below a structural depth of four. Detailed documents may have numbered statements down to a depth of nine. A document that is well organized and maintains a consistent level of detail will have a pyramidal shape (few numbered statements at level 1 and each lower level having more numbered statements than the level above it). Documents that have an hour-glass shape (many numbered statements at high levels, few at mid levels and many at lower levels) are usually those that contain a large amount of introductory and administrative information. Diamond shaped documents (a pyramid followed by decreasing statement counts at levels below the pyramid) indicate that subjects introduced at the higher levels are probably addressed at different levels of detail.

- SPECIFICATION DEPTH is a count of the number of imperatives at each level of the document. These numbers also include the count of lower level list items that are introduced at a higher level by an imperative that is followed by a continuance. This structure has the same implications as the numbering structure. However, it is significant because it reflects the structure of the requirements as opposed to that of the document. Differences between the shape of the numbering and specification structure are an indication of the amount and location of background and/or introductory information is included in the document. The ratio of total for SPECIFICATION STRUCTURE to total lines of text is an indication of how concise the document is in specifying requirements.

The application of this information is still under investigation, and initial results from Project X are interesting. Figure 3 depicts expected structure versus actual structure of the Specification and Design requirement documents. The project data suggests the Specification requirements may have been overly defined, therefore artificially constraining the design and its expansion. The structure of the imperative levels in the Design document reinforces this observation, indicating little expansion where extensive expansion is expected.

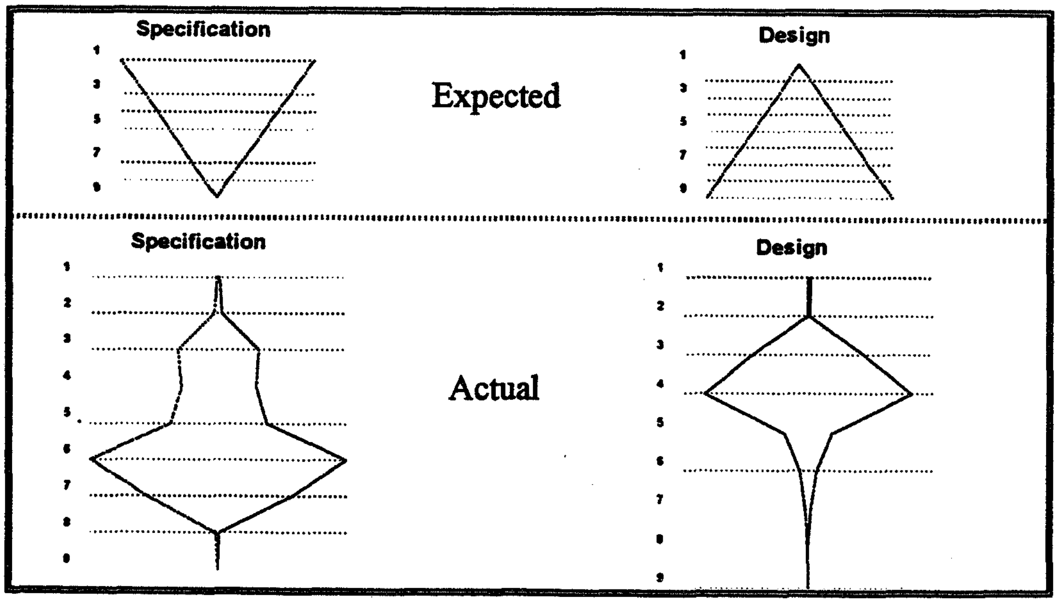


Figure 3: Document Depth at Which Imperatives are Located

3. Requirement Metrics

This section of the paper focuses on the application of metrics available through the use of a requirements management CASE tool. These metrics assist project managers and quality assurance engineers to identify the risks of insuring that the completed software system contains the functionality specified by the requirements. There are no published or industry standard guidelines for these metrics: intuitive interpretations, based on experience and supported by project feedback, are used in this paper. Project management has reacted favorably to the metrics and has used the analysis results to mitigate certain perceived risks. The SATC continues working on methods to mathematically validate the intuitive guidelines so that the requirement metrics and their interpretation are applicable to an ever increasing variety of software development applications. Three areas of requirement metrics will be discussed: Stability Over Time Per Requirement Design Level, Stability Over Time By Project Build, Expansion From Specification To Design Level.

3.1 Requirement Stability Over Time per Requirement Design Level

Requirements are developed and baselined at major reviews during the system development life cycle. At these milestone reviews, documents containing the requirements are reviewed and commented upon. After resolution of the comments, the requirement documents are baselined and put under configuration control. Ideally, the rate of change in each level of requirements should decrease as a milestone review approaches. Figure 4 shows the count of requirements at each level during the 6 month period starting at Preliminary Design Level (PDR) (through Critical Design Review (CDR) As expected, the Top Level and Specification requirements remained stable during this six month period. The Intermediate Specification and Design Level documents both stabilized prior to CDR.

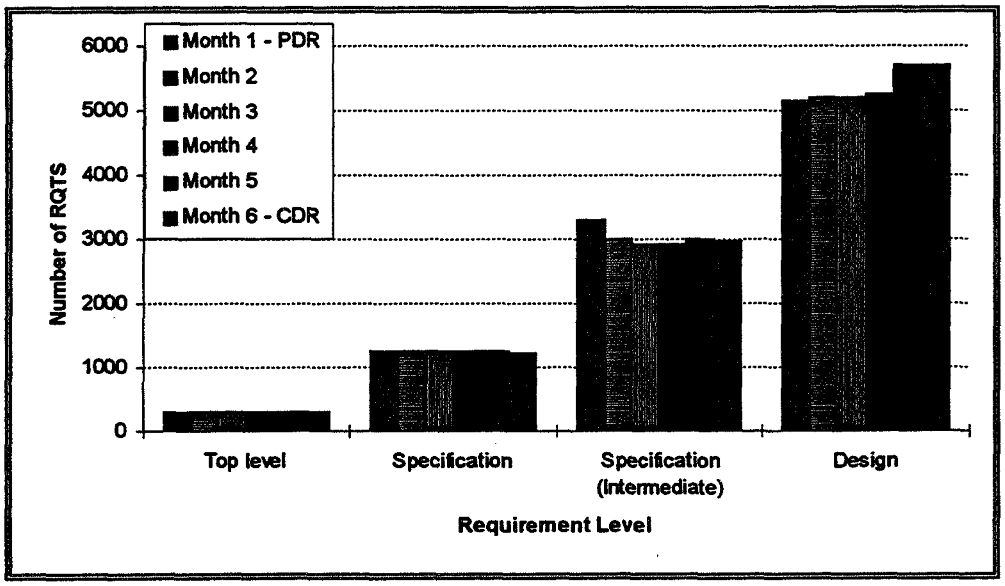


Figure 4: Requirement Count by Document Level

3.2 Requirement Stability Over Time By Project Build

As stated earlier, this system is implemented in three builds with the Specification and Design requirements allocated to each of these builds. One of the purposes of a multiple build development effort is to minimize the implementation risk associated with any one build. This insures that no single build implements an inordinate number of requirements. Figure 5 shows the counts of the Design Requirements (Figure 4) for Build 1 and Build 2.

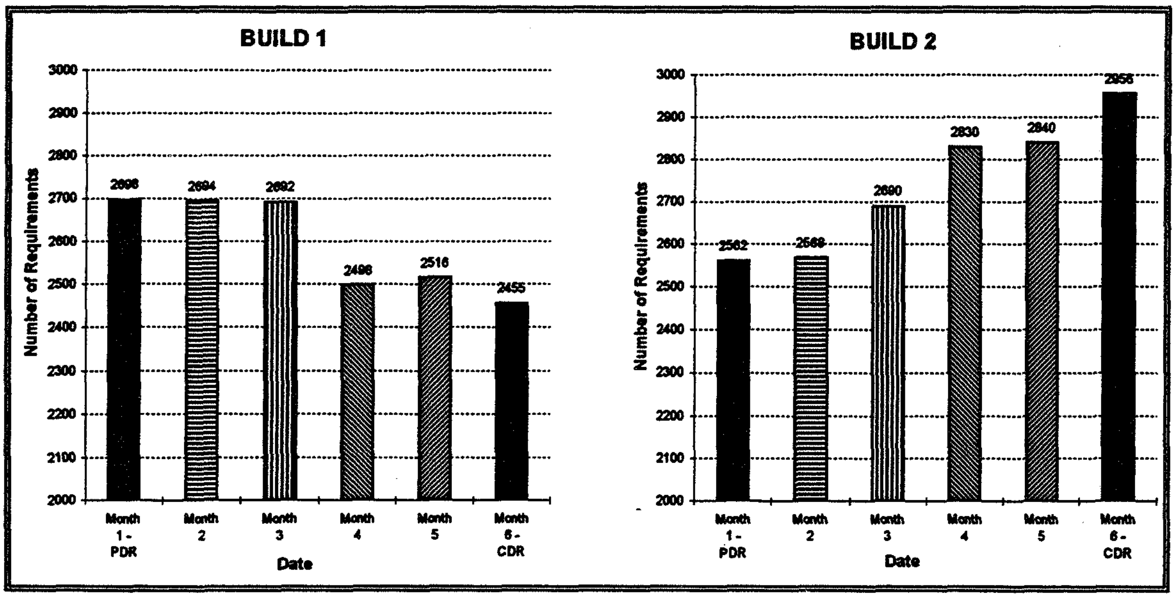


Figure 5: Design Requirement Allocation by Build

This is a different picture of the requirements stability, showing a shift in the number of requirements from Build 1 to Build 2; and indicating a potential risk to the schedule of Build 2 which should be closely monitored.

3.3 Requirement Detail Expansion

In addition to requirement stability, the expansion of the upper level requirements to more detailed levels generates potential project risk. Figure 6 shows the number of requirements of the Detail level referencing the number of requirements in the Intermediate Specification Level. The tails on the expected curve in the upper right indicate a scattering of upper level requirements referenced either by very few or very many detailed requirements; however, the majority of requirements will have multiple references and result in a bell-shaped curve..

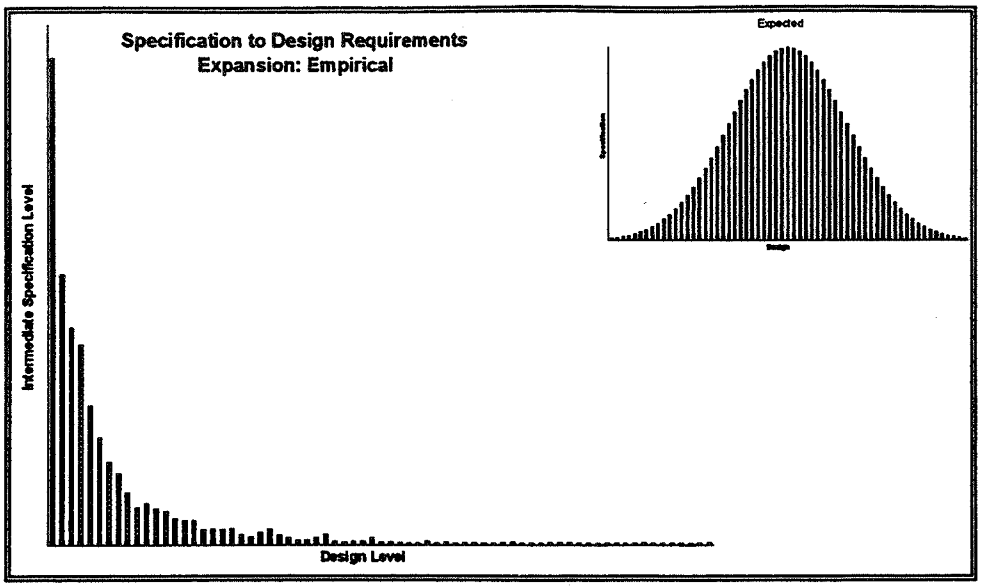


Figure 6: Requirement Expansion

Project data however, does not match the expected; there is a high number of Intermediate Specification requirements that are referenced by only one or very few Detail requirements (left hand side of the graph) while other requirements have high numbers of multiple references (right hand side of graph). As an example, the Intermediate Specification requirement "The system shall have a database." is linked to 200 Design level requirements. The shape of the curve in Figure 6 indicates that the Design analysis is incomplete with the specific requirements are not adequately decomposed, thus suggesting that requirements were copied with neither analysis nor expansion into detail for the implementation phase.

4. Conclusions

Based on the work done to date, four conclusions can be reached:

- Requirement metrics assist in identifying potential project risks
- Multiple metrics are needed for comprehensive evaluation
- Evaluation of requirement text can yield risk information very early in the life cycle
- Metric collection is cheaper, faster and more reliable with requirement management tools

Using automated tools to track requirements has opened the door to deriving metrics for characterizing requirement text , stability and expansion rate. Tracking and correlating test cases and test results to individual requirements within a database is essential for viewing relationships not otherwise available.. The use of an automated requirements database allows the metrics program to generate metrics for best insight into the requirements and test case interplay . The metrics presented in this paper are the result of much research into data use and pictorial display; however, all results have been used by project management to successfully identify and manage risks.

REFERENCES

Brooks, Frederick P. Jr., No Silver Bullet: Essence and accidents of software engineering, *IEEE Computer, vol. 15, no. 1,* April 1987, pp. 10-18.

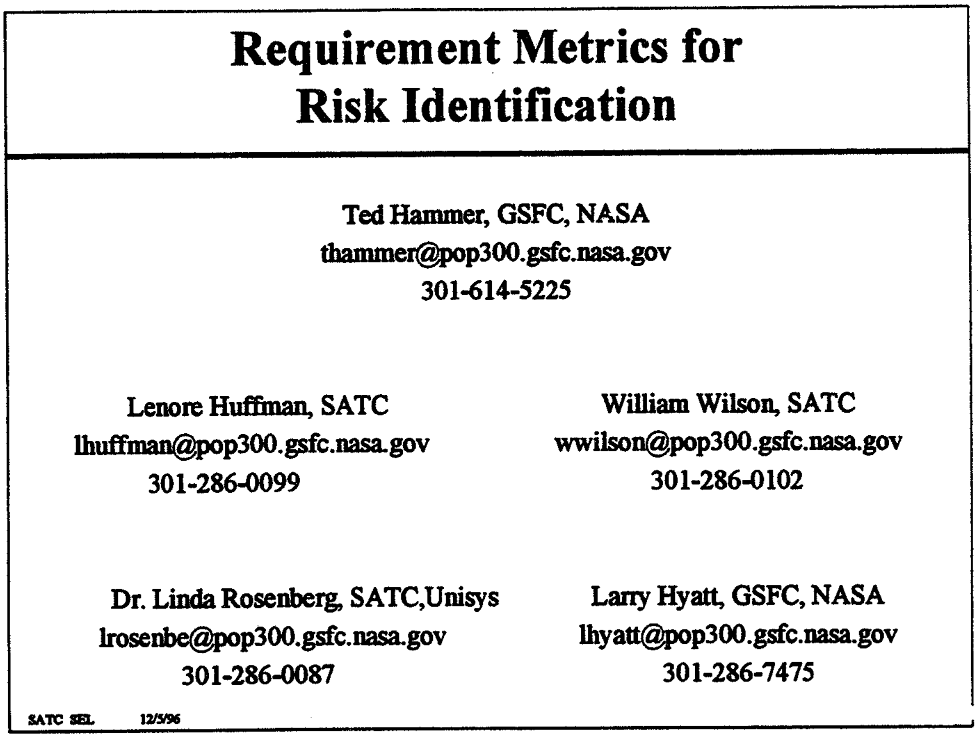
DOD MIL-STD-490A, Specification Practices, June 4, 1985.

IEEE Std 830-1993, Recommended Practice for Software Requirements Specifications, December 2, 1993.

Kitchenham, Barbara, Pfleeger, Shari Lawrence, Software Quality: The Elusive Target, *IEEE Software, Vol. 13, No. 1,* January 1996, pp. 12-21.

Stokes, David Alan, Requirements Analysis, *Computer Weekly Software Engineer's Reference Book*, 1991, pp. 16/3-16/21.

Wilson, William, Rosenberg, Linda, Hyatt, Lawrence, Automated Quality analysis of Natural Language Requirement Specifications, Fourteen Annual Pacific Northwest Software Quality Conference, October, 1996.



Requirement Metrics for Risk Identification

Ted Hammer, GSFC, NASA
thammer@pop300.gsfc.nasa.gov
301-614-5225

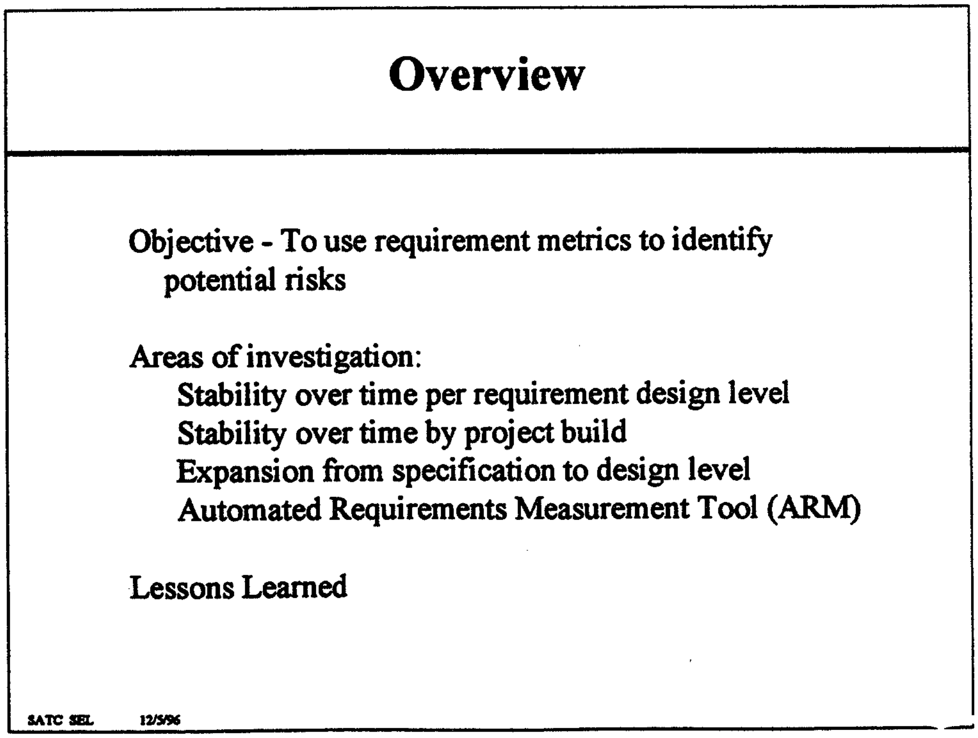
Lenore Huffman, SATC
lhuffman@pop300.gsfc.nasa.gov
301-286-0099

William Wilson, SATC
wwilson@pop300.gsfc.nasa.gov
301-286-0102

Dr. Linda Rosenberg, SATC,Unisys
lrosenbe@pop300.gsfc.nasa.gov
301-286-0087

Larry Hyatt, GSFC, NASA
lhyatt@pop300.gsfc.nasa.gov
301-286-7475

SATC SEL 12/5/96



Overview

Objective - To use requirement metrics to identify potential risks

Areas of investigation:
- Stability over time per requirement design level
- Stability over time by project build
- Expansion from specification to design level
- Automated Requirements Measurement Tool (ARM)

Lessons Learned

SATC SEL 12/5/96

Software Assurance Technology Center (SATC)

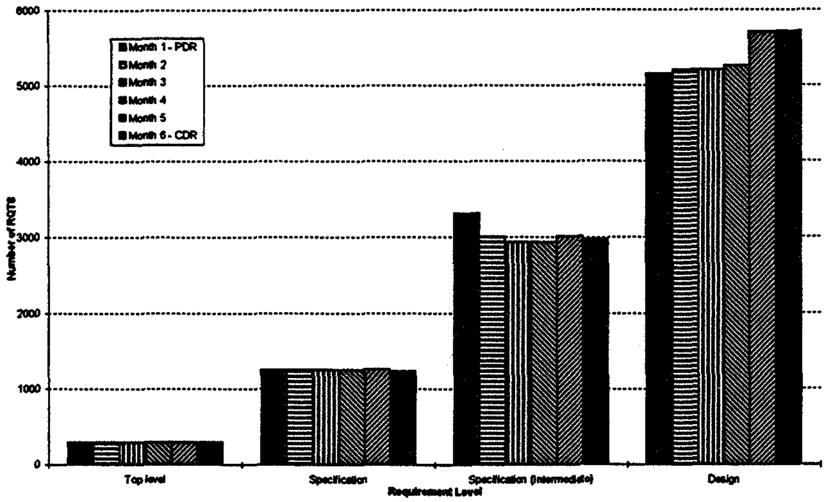
Primary Responsibilities:

- Software Guidebooks, Standards
- Metrics Research and Development
- Assurance Tools and Techniques
- Outreach / Project Support
  - http://satc.gsfc.nasa.gov/homepage.html

Supported primarily by NASA Software Technology Division, WV

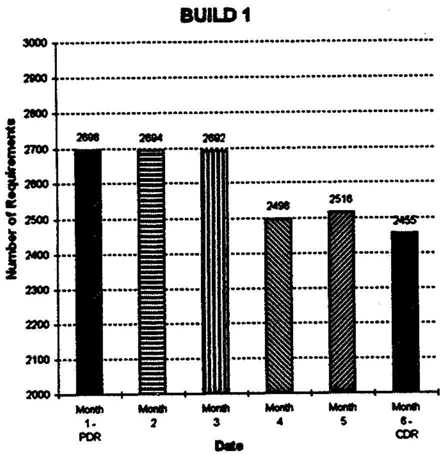
SATC SEL 12/5/96

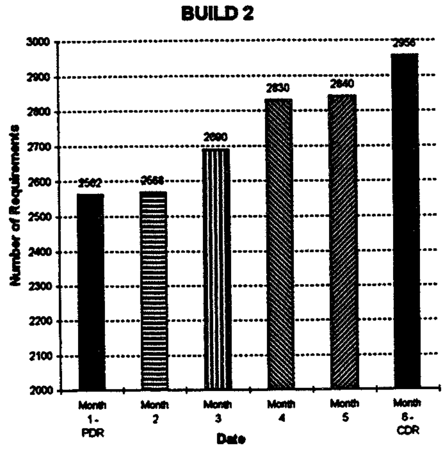
Requirement Count by Document Detail Level



SATC SEL 12/5/96

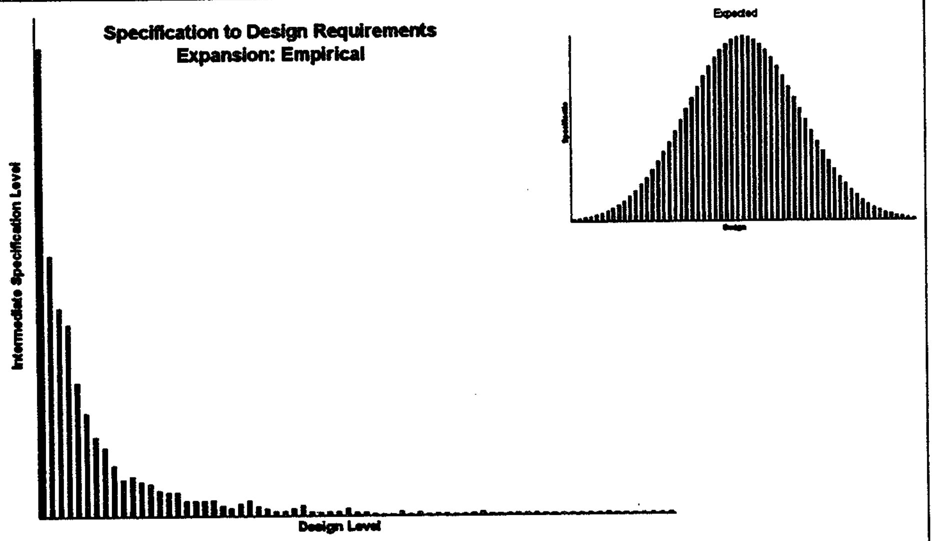
Requirement Stability By Build



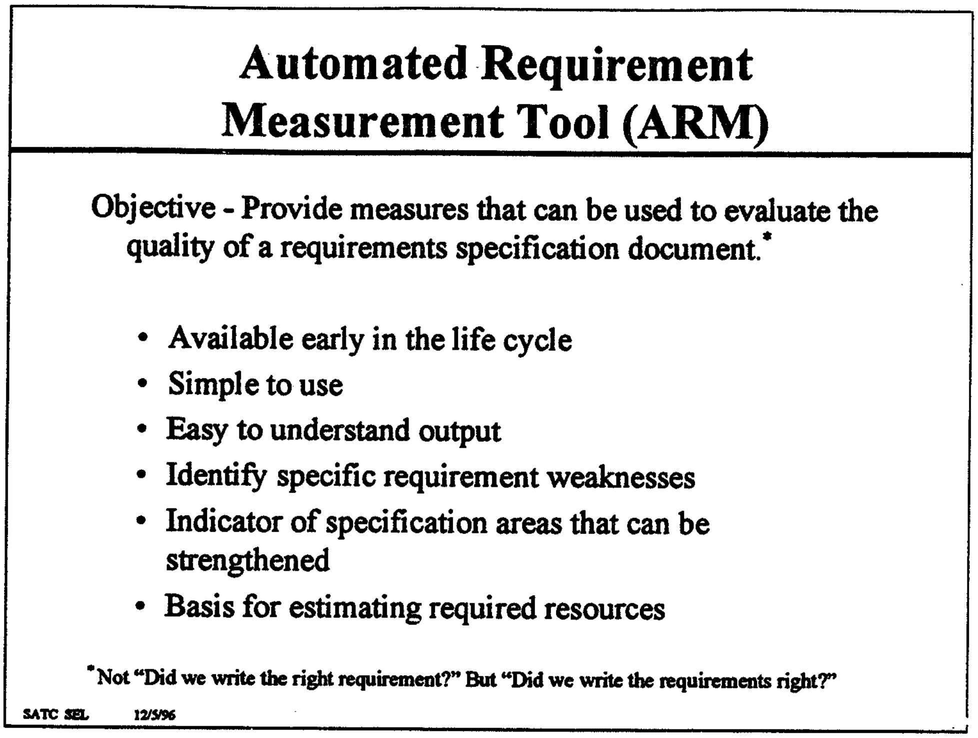


SATC SEL 12/5/96

Requirement Expansion



SATC SEL 12/5/96



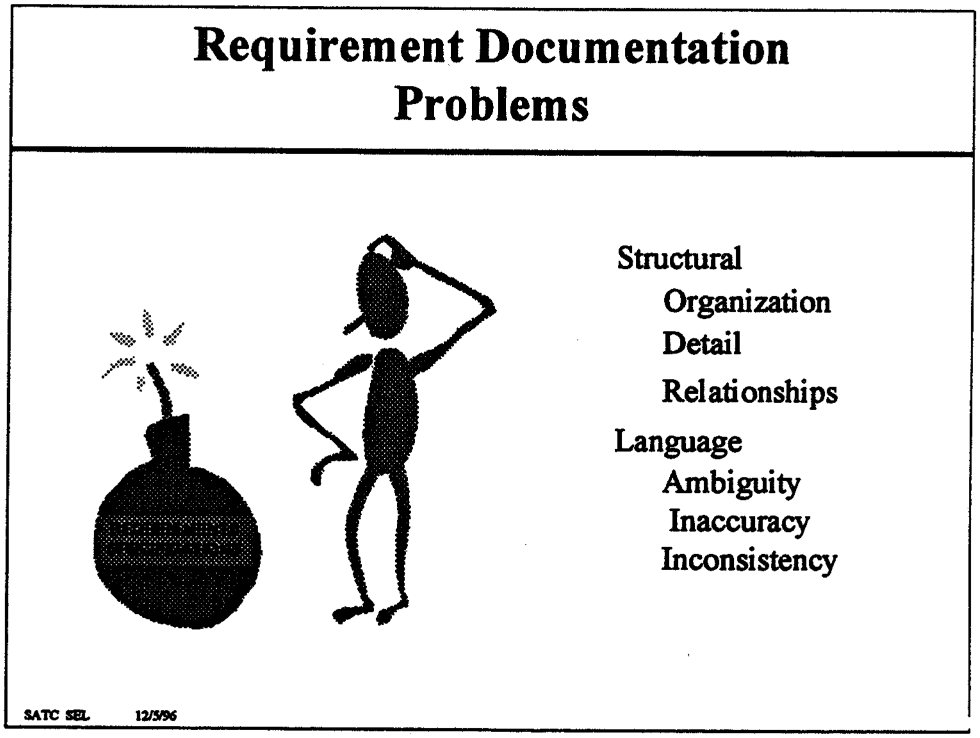
Automated Requirement Measurement Tool (ARM)

Objective - Provide measures that can be used to evaluate the quality of a requirements specification document.*

- Available early in the life cycle
- Simple to use
- Easy to understand output
- Identify specific requirement weaknesses
- Indicator of specification areas that can be strengthened
- Basis for estimating required resources

*Not "Did we write the right requirement?" But "Did we write the requirements right?"

SATC SEL 12/5/96

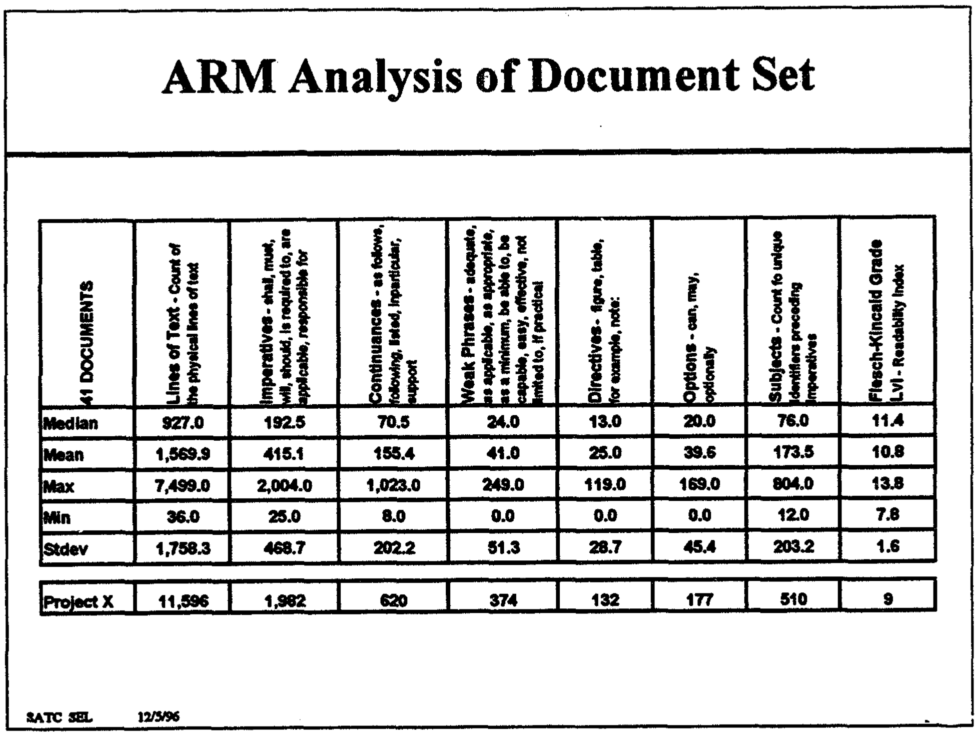


Requirement Documentation Problems

Structural
Organization
Detail
Relationships

Language
Ambiguity
Inaccuracy
Inconsistency

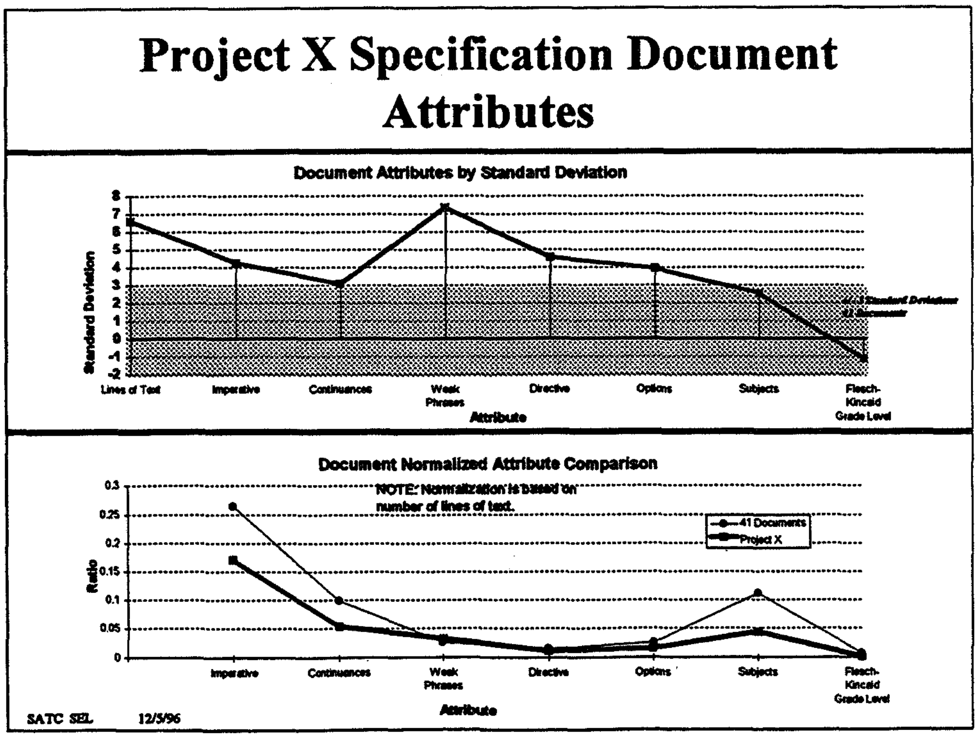
SATC SEL 12/5/96



ARM Analysis of Document Set

41 DOCUMENTS	Lines of Text - Count of the physical lines of text	Imperatives - shall, must, will, should, is required to, are applicable, responsible for	Continuances - as follows, following, listed, inparticular, support	Weak Phrases - adequate, as applicable, as appropriate, as a minimum, be able to, be capable, easy, effective, not limited to, if practical	Directives - figure, table, for example, note:	Options - can, may, optionally	Subjects - Count fo unique identifiers preceding imperatives	Flesch-Kincaid Grade Lvl - Readability Index
Median	927.0	192.5	70.5	24.0	13.0	20.0	76.0	11.4
Mean	1,569.9	415.1	155.4	41.0	25.0	39.6	173.5	10.8
Max	7,499.0	2,004.0	1,023.0	249.0	119.0	169.0	804.0	13.8
Min	36.0	25.0	8.0	0.0	0.0	0.0	12.0	7.8
Stdev	1,758.3	468.7	202.2	51.3	28.7	45.4	203.2	1.6
Project X	11,596	1,982	620	374	132	177	510	9

SATC SEL 12/5/96



Project X Specification Document Attributes

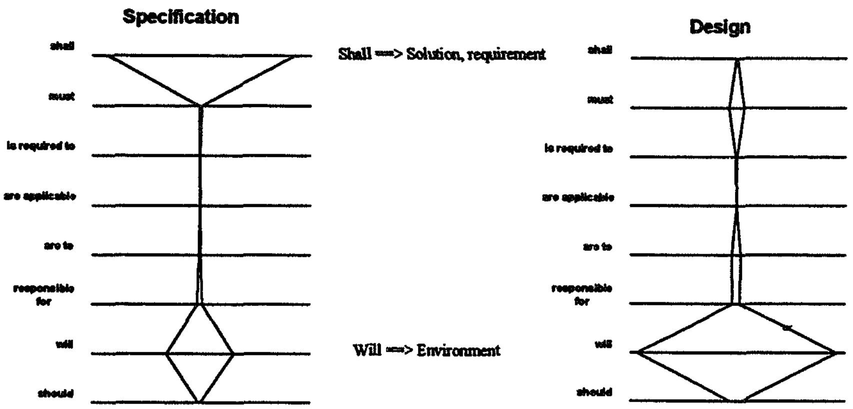
Document Attributes by Standard Deviation

Document Normalized Attribute Comparison

NOTE: Normalization is based on number of lines of text.

SATC SEL 12/5/96

Imperatives

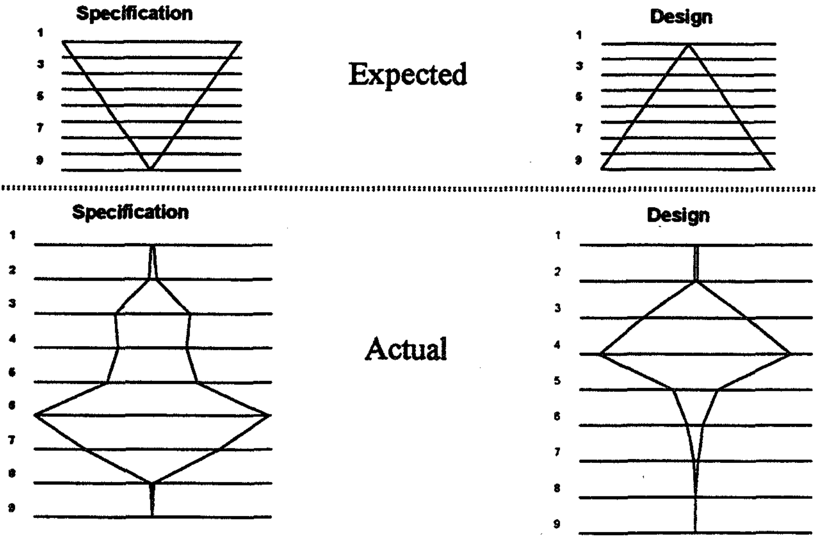


Shall ==> Solution, requirement

Will ==> Environment

SATC SEL 12/5/96

Structure
Level at Which Imperative Occurs



Expected

Actual

SATC SEL 12/5/96

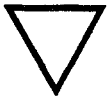
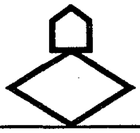
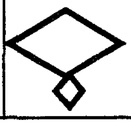
Lessons Learned

- Requirement metrics assist in identifying potential project risks
- Multiple metrics are needed for comprehensive evaluation
- Evaluation of requirement text can yield risk information very early in the life cycle
- Metric collection is cheaper, faster and more reliable with requirement management tools

SATC SEL 12/5/96

Summary

Project Phases - Graphical Representation

PHASE ARTIFACT ➤ TYPE OF EVIDENCE ▼	RQT Stability	RQT Expansion	Structure Specification Doc	Structure Design Doc	Tests per RQT	NCR's
THEORETICAL*						7-12 / 1K Source
EMPIRICAL Project X						. . .

Continued research focus

SATC SEL 12/5/96

516-61
415662
2L03E
360762
p 24

Applying the SCR Requirements Specification Method to Practical Systems: A Case Study[§]

Ramesh Bharadwaj and Connie Heitmeyer
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375-5320
E-mail: {ramesh,heitmeyer}@itd.nrl.navy.mil

1 Introduction

Studies have shown that the majority of errors in software systems are due to incorrect requirements specifications. The root cause of many requirements errors is the imprecision and ambiguity that arise because the software requirements are expressed in natural language. An effective way to reduce such errors is to express requirements in a formal notation. For a number of years, researchers at the Naval Research Laboratory (NRL) have been working on a formal method based on *tables* to specify the requirements of practical systems [2, 11]. Known as the Software Cost Reduction (SCR) method, this approach was originally formulated to document the requirements of the Operational Flight Program (OFP) for the U.S. Navy's A-7 aircraft [2]. Since SCR's introduction more than a decade ago, many industrial organizations, including Lockheed, Grumman, and Ontario Hydro, have used SCR to specify requirements. Recently, NRL has developed both a formal state machine model [12, 14] to define the SCR semantics and a set of software tools to support analysis and validation of SCR requirements specifications [10]. The tools support consistency and completeness checking, simulation, and model checking.

To evalute the SCR method and toolset, we recently used SCR to produce a black box requirements specification of a simplified mode control panel for the Boeing 737 autopilot. Beginning with the English language description of the system presented in [4], we represented the environmental quantities that the computer system monitors (e.g., the pilot switches, dials, and sensors) and the environmental quantities that the computer system controls (i.e., the individual displays) as monitored and controlled variables. We then used these variables and the SCR tabular notation to specify the requirements of the mode control panel. The heart of the specification is the relation REQ, the required relation between the monitored and controlled variables [20].

In this paper, we use the autopilot mode control panel as an example for comparing and contrasting the SCR approach to requirements specification and analysis with the approach used in [4]. The latter approach uses the formal language of SRI's Prototype Verification System (PVS) [17] to represent the requirements of the mode control panel and then applies the automated reasoning provided by PVS to analyze the specification. Formulating the requirements specification

[§] This work was supported by the Office of Naval Research.

for the mode control panel in SCR exposed a number of problems, including a missing input event, an incorrect assumption about the environment, and a misinterpretation of the prose description. We also discovered that because parts of the PVS specification are highly abstract, certain key aspects of the system's requirements are omitted. In contrast, the SCR approach makes explicit many important questions about the required behavior of the mode control panel. We conclude with a discussion of general issues such as the appropriate level of abstraction for documenting requirements, the choice of notation, the kinds of analyses that can be done on the specification, the relation between different kinds of analyses, and the role of tool support. Appendix B contains the complete SCR requirements specification of the mode control panel.

2 Motivation and Background

It is widely acknowledged that requirements are a major source of errors during the development of large software systems [1, 9, 16]. For example, studies by Lutz [16] have shown that functional and interface requirements were the source of a majority of safety-related software errors in NASA's Voyager and Galileo spacecrafts. There is no doubt that getting a complete and consistent characterization of software requirements is inherently hard. However, there are failings in the software development process, including the requirements process, that can be rectified by improved practice [8]. A disciplined and rigorous approach to the analysis and specification of software requirements can address many difficulties that result from such failings.

The goal of the requirements phase is to create a document, the Software Requirements Specification (SRS), to precisely describe the problem to be solved and to accurately characterize the set of acceptable solutions to the problem. The effectiveness of the requirements phase is determined by the extent to which the SRS is precise, unambiguous and consistent (i.e., its correctness), whether it captures all the results of the analysis (i.e., its completeness), and its useability. The useability criteria are ease of change (i.e., its modifiability), whether the notation is understandable both by customers as well as the developers (i.e., its readability), its organization for easy reference and review (for instance, one should quickly be able to find answers to specific questions about the requirements), and organization for ease of change. In addition, the underlying conceptual model and notation of the SRS should support formal analyses such as *validation* (to ensure that the specification describes the intended requirements), and *verification* (which establishes that the specification satisfies critical properties of interest). Finally, the method should provide guidelines that support decisions on organization and modification of the SRS. By sufficiently constraining the underlying semantic model, these guidelines ensure that the quality of the SRS does not depend too much on the level of expertise of its writer(s).

2.1 The SCR Method

Unlike traditional research on requirements, which concentrates on the *requirements analysis process*, the focus of the SCR work at the Naval Research Laboratory is on issues that influence the creation and maintenance of the SRS. By identifying desirable properties of an SRS, the SCR project has developed a set of guidelines for writing the SRS [11, 8]. These guidelines include *separation of concerns*, *information hiding*, and the use of a readable yet *formal notation*. For example, the guideline *separation of concerns* supports useability, modifiability, and verifiability of the SRS. Moreover, the notation supported by the SCR method is designed to be understandable both by customers as well as software developers. Underlying the notation is a mathematical model which supports completeness and consistency checking, validation, test case generation, and formal verification.

To support the SCR method, NRL has developed a set of software tools for analysis and validation of SCR requirements specifications [10, 13]. The tools include a *specification editor* for creating and modifying the specifications, a *simulator* for symbolic execution, and tools for formal analysis. The latter include a *consistency checker* which uncovers application-independent errors such as syntax and type errors, missing cases, and unwanted nondeterminism, and a *verifier* which checks a specification for critical application-specific properties.

2.2 PVS

PVS (Prototype Verification System) [17] is an environment for specification and verification developed at SRI International. The PVS system is built around a highly expressive specification language. The system has a number of predefined theories, and comes with a very effective interactive theorem prover in which most of the low-level proof steps are automated. The PVS specification language is based on higher-order logic with a richly expressive type system. The PVS prover consists of a powerful collection of inference steps which include arithmetic and equality decision procedures, automatic rewriting, and boolean simiplification. PVS has been applied to a number of practical problems [4, 5, 21]. Many organizations, including NASA, have used the PVS specification language for documenting software requirements.

3 Comparison of PVS with the SCR method

In this section, we address some of the strengths and limitations of using PVS, and compare the PVS approach to the SCR method. We base our comparison on the assumption that a notation (and associated tools) should support the following process, which may be thought of as an idealization of a real-world process for requirements analysis [19].

1. **SRS Creation:** The results of problem analysis are captured in the SRS, using a formal notation.

2. **SRS Checking:** The SRS is checked for proper syntax, type correctness, consistency, completeness, and other application-independent properties, using an automated checker.

3. **SRS Validation:** The goal of this phase is to ensure that the SRS captures the customers' intent. This is achieved by symbolically executing the SRS using a simulator.

4. **SRS Verification:** This phase verifies that certain crucial application specific properties, such as safety and security properties, hold for the SRS. Verification is carried out by using an interactive theorem prover or by "lightweight" analysis tools such as model checkers.

3.1 SRS Creation

The choice of notation, and availability of guidelines to support decisions on SRS organization and modification, are factors which influence this phase. A simpler, more restrictive notation is preferable to a more powerful, expressive one. In addition to ease of use, a restricted semantic model can provide guidelines for creating and organizing the SRS. A well-designed notation will help even novices create good specifications.

The PVS system is built around a highly expressive specification language. However, most developers, being unfamiliar with higher-order logic (the underlying formalism of the PVS specification notation), lambda expressions, higher-order functions and quantification, etc, find the notation hard to use. It has also been our experience that the expressive power of higher-order logic is seldom required for requirements specification of most practical systems. The organizing unit for PVS specifications is the "Theory". The PVS language lacks structures to support readability and ease of change. It is very hard for novices to create good PVS specifications. For example, it has been observed by Young [22] that the quality of specifications in PVS depends to a large extent on the expertise of the specification writer.

The SCR method is suitable for embedded, real-time systems, i.e., for systems that sense and control quantities in their environment [20]. The SCR method includes a systematic approach for capturing requirements [11, 15, 6], and is based on a tabular notation which has a formal mathematical basis [12, 13, 14]. The SCR notation, having been tailored to a specific class of problems, sacrifices generality for ease of use and improved support for analysis. Most engineers find the tabular notation easy to use and understand. Also, tables afford a natural organization which permits independent construction, review, modification, and analysis of smaller parts of a large requirements specification.

It has been observed that in comparison to graphical notations and (structured) text, tabular notations scale very well to large problems. According to Parnas, the specification of the shutdown system for the Darlington Nuclear Power Plant [18] weighed more than 20 kilograms on paper. In our own experience, we have come across examples of SCR requirements specifications for practical systems (e.g., the OFP for the C-130J aircraft [7]) containing more than a thousand tables.

3.2 SRS Checking

In addition to checks for incorrect syntax, the PVS language has a rich type system which supports rigorous typechecking. The type system of PVS is undecidable, which means that typechecking cannot be completely automated. In most situations, the PVS typechecker will generate proof obligations which have to be proved using the interactive prover. Such proofs amount to a very strong consistency check on some aspects of the specification.

The consistency and completeness checker of the SCR toolset verifies application-independent properties derived automatically from the requirements model. These checks ensure that a specification is well-formed by identifying syntax and type errors, incompleteness, missing initial values, unreachable modes, and circular definitions. The tool also identifies missing cases and undesirable nondeterminism. All these checks are carried out automatically.

3.3 SRS Validation

PVS does not support validation.

The tabular notation of SCR supports validation by inspection and simulation. Most domain experts find this notation easy to read and review. For example, Parnas [18] observes that the utility of the tabular notation was evident during the formal review of the Darlington specification. During the review, each "case" and its associated subcases could be reviewed individually and independently of other "cases". The tabular notation also forces one to consider all possible scenarios. Further, we show in [3] that theorems that are true of certain fragments of an SCR requirements specification also hold for the whole specification.

The simulator in the SCR toolset performs symbolic execution of the underlying state machine model, which allows users to assess system behavior in specific "use cases" directly from the requirements specification. The simulator can expose problems — such as missing requirements and incorrectly stated requirements — that cannot be detected by verification techniques.

3.4 SRS Verification

Using PVS, one can establish, by interactive theorem proving, properties that are deemed to be true of a requirements specification. However, few practitioners have the mathematical sophistication required to carry out such proofs. The state-of-the-art theorem prover of PVS does ameliorate the problem by including powerful decision procedures that automate parts of a proof that would otherwise require user guidance. Very often, a property *will not* hold for a requirements specification. In such a case, either the formulation of the property is incorrect, or the specification is wrong (or both). Proper diagnosis and user feedback are therefore very important to help correct the problem. Theorem provers provide very little help in such situations because theorem proving is incomplete; i.e., if one is unable to prove a theorem using a theorem prover, then all one can conclude is that the theorem prover failed to find a proof (the theorem may be true). On the other hand, methods such as model checking are complete — if a model checker reports that a theorem is false, it is false. Additionally, most model checkers will provide a *counterexample* that falsifies the theorem. PVS does support model checking for a limited subset of the language, but provides no counterexample.

The SCR toolset supports proof of safety properties of a requirements specification using state exploration based model checking [3]. One of the main design goals of our toolset is to provide proper error diagnosis by generating understandable counterexamples for user feedback. Future plans include support for other forms of model checking and automatic theorem proving. Since the underlying model of the SCR notation is a state machine, several other verification activities can be supported. For instance, we plan to automatically generate test-cases from an SCR specification, to assist in black-box testing of implementations. In certain limited contexts, it should also be possible to automatically generate code directly from an SCR requirements specification.

4 The Autopilot Requirements Specification

To illustrate the SCR method, we consider a simplified mode control panel for the Boeing 737 autopilot as discussed in [4]. The mode control panel for the autopilot is shown in *Figure 1.*

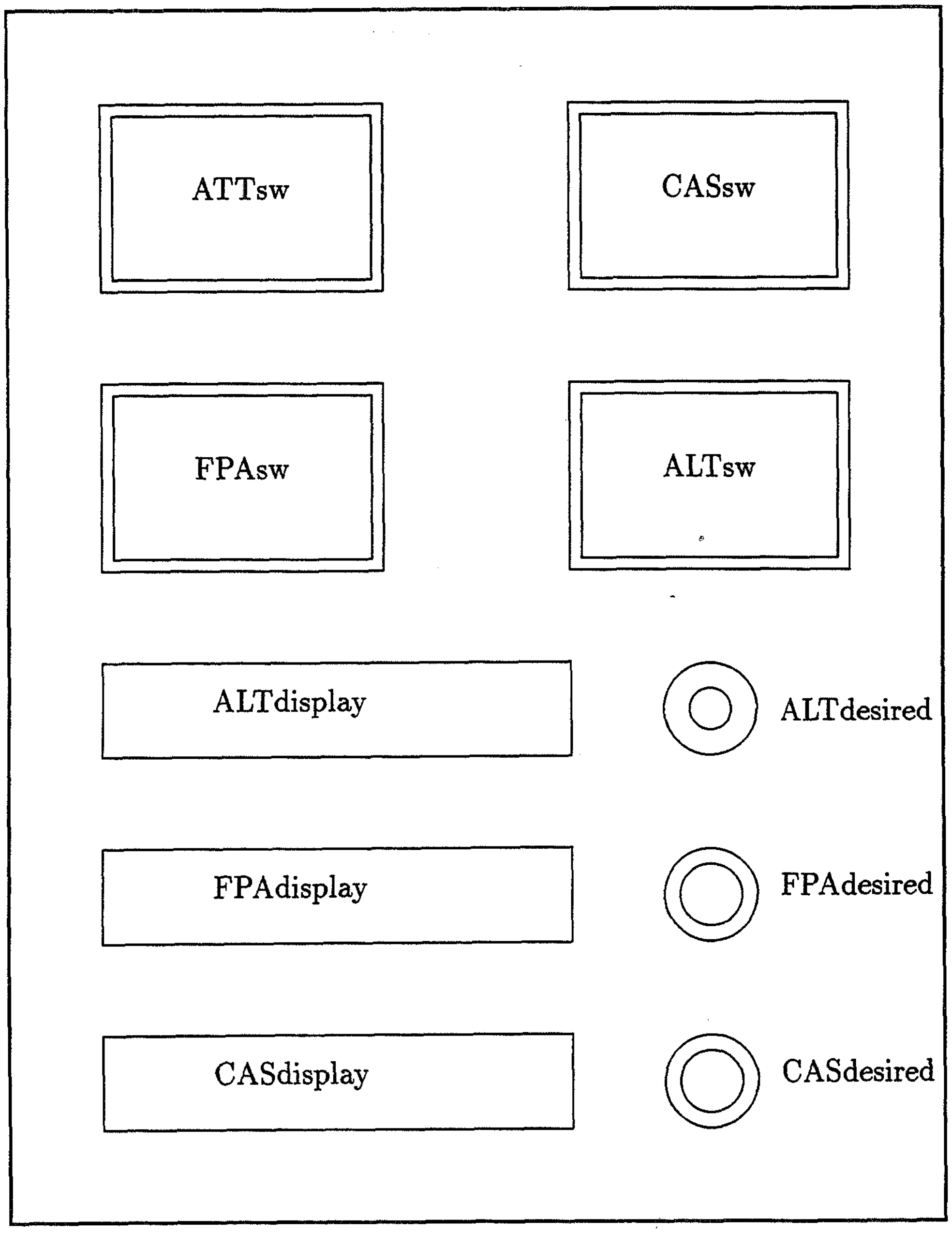


Figure 1: Mode Control Panel

The system monitors the aircraft's altitude (ALT), flight path angle (FPA) and calibrated air speed (CAS). The panel includes three displays which show the current values for altitude, flight path angle, and airspeed of the aircraft. The pilot may enter a new value into a display by "dialing-in" the value using one of three knobs next to the displays. The pilot engages or disengages the autopilot by pressing one of four buttons on the panel. Appendix A contains a description of the system in English prose (adapted from [4]). Below, we informally present the steps taken to document the requirements using the SCR notation.

In SCR, the required system behavior is described by REQ, the required relation between *monitored variables*, environmental quantities that the system monitors, and *controlled variables*, environmental quantities that the system controls [20]. To specify this relation concisely, the SCR approach uses four constructs – modes, terms, conditions, and events. A *mode class* is a variable whose values are *system modes* (or simply *modes*), while a *term* is any function of monitored variables, modes, or other terms. A *condition* is a predicate defined on one or more system entities (an *entity* is a monitored or controlled variable, mode class, or term). An *event* occurs when the value of any system entity changes. The notation "@T(c) WHEN d" denotes a *conditioned event*, defined as

$$\texttt{@T(c) WHEN d} \overset{\text{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed condition c is evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state. The notation "@F(c)" denotes the event @T(NOT c). The environment may change a monitored quantity, causing an *input event*. In response, the system changes controlled quantities and updates terms and mode classes.

We begin by identifying the monitored quantities, i.e., the environmental quantities that the autopilot system monitors, and denote them by corresponding *monitored variables*. We use the prefix "m" for all monitored variable names. Each monitored variable is of a certain *type*, which specifies the range of values that may be assigned to that variable. The autopilot system monitors the actual altitude (denoted by monitored variable mALTactual), the actual flight path angle (mFPAactual), and the actual calibrated air speed (mCASactual). We assume these variables to range over the integers. Switches ALTsw, ATTsw, CASsw, and FPAsw are denoted respectively by mALTsw, mATTsw, mCASsw, and mFPAsw. These monitored variables may take on one of the values from the set {on, off}. Finally, knobs ALTdesired, CASdesired, and FPAdesired are denoted by monitored variables mALTdesired, mCASdesired, and mFPAdesired respectively, which range over the integers.

We then identify the controlled quantities, i.e., the environmental quantities that the autopilot system controls, and denote them by corresponding *controlled variables*. We use the prefix "c" for all controlled variable names. Just as for monitored variables, we assign a *type* to each controlled variable. For simplicity of exposition we shall, as in [4], only model the mode-control panel itself, and not the commands that will be sent out to the flight-control computer. The three controlled quantities of the mode control panel are ALTdisplay, FPAdisplay, and CASdisplay, which we denote respectively by cALTdisplay, cFPAdisplay, and cCASdisplay. We assume these values to range over the integers.

We model the primary modes of the mode-control panel by the *modeclass* Status, denoted by variable mcStatus. The variable can take on any value in the set {ALTmode, ATTmode, FPAmode}. The altitude engaged mode being "armed" is denoted by a boolean *term* variable tARMED (we use the prefix "t" for terms). If tARMED is *true*, then mcStatus should be FPAmode. The previous sentence is an example of a property of the specification which we may later want to prove. We also define a boolean valued *term* tCASmode, to model the system being in the *calibrated air speed* mode. By describing the status of the mode-control panel in this manner, we have ensured that the following sentences in the prose requirements are trivially satisfied:

1. Only one of the three modes ALTmode, ATTmode, or FPAmode can be engaged at any time.
2. One of the three modes, ATTmode, FPAmode, or ALTmode should be engaged at all times.
3. Engaging any of the three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.
4. The mode CASmode can be engaged at the same time as any of the other modes.

We define three boolean valued terms tALTpresel, tCASpresel, and tFPApresel to denote whether the corresponding quantity has been pre-selected by dialing in a new value using one of the three knobs. Finally, we define a boolean term tNear to denote the predicate mALTdesired − mALTactual ≤ 1200.

The behavior of mode class mcStatus is specified in a *mode transition table*. In the following, the expression CHANGED(x) denotes the event "variable x has changed". The table defines all events that change the value of the mode class mcStatus. For example, the first row of the table states, "If mcStatus is ALTmode, and mATTsw is switched on, or the setting of knob mALTdesired is changed, then mcStatus changes to ATTmode." Events that do not change the value of the mode class are omitted from the table.

Source Mode	Events	Destination Mode
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode
ALTmode	@T(mFPAsw = on)	FPAmode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
ATTmode	@T(mFPAsw = on) OR @T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAmode
FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear) OR @T(tNear) WHEN tARMED	ALTmode
FPAmode	@T(mATTsw = on) OR @T(mFPAsw = on) OR CHANGED(mALTdesired) WHEN tARMED	ATTmode

Each row in the mode transition table above corresponds to certain sentences in the prose requirements. We describe this correspondence below. Here, "paragraph *x*" refers to the numbered paragraph *x* of the prose requirements in Appendix A.

Row 1. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., pressing ATTsw should engage ATTmode **OR** *If the pilot dials in a new altitude while* ALTmode *is engaged, then* ALTmode *is disengaged and* ATTmode *is engaged* (paragraph 7).

Row 2. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., by pressing FPAsw the pilot engages FPAmode.

Row 3. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., pressing ALTsw engages ALTmode. *However, the altitude must be pre-selected before* ALTsw *is pressed* (paragraph 4). *If the pilot dials an altitude that is more than* 1,200 *feet above* ALTactual *and then presses* ALTsw, *then* ALTmode *will not directly engage* (paragraph 3).

Row 4. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., by pressing FPAsw the pilot engages FPAmode **OR** *If the pilot dials into* ALTdesired *an altitude that is more than* 1,200 *feet above* ALTactual *and then presses* ALTsw, *then* ALTmode *will not directly engage. Instead, the altitude engage mode will change to "armed" and* FPAmode *is engaged* (paragraph 3).

Row 5. The situation described for row (3) above **OR** *Instead, the altitude engage mode will change to "armed" and* FPAmode *is engaged. [...]* FPAmode *will remain engaged until the aircraft is within* 1,200 *feet of* ALTactual, *then* ALTmode *is automatically engaged* (paragraph 3).

Row 6. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., by pressing mATTsw the system enters ATTmode **OR** FPAsw *toggles on and off every time it is pressed.* (paragraph 5) **OR** *If the pilot dials in a new altitude while the altitude engage mode is "armed" then* ATTmode *is engaged. [...]* FPAmode *should be disengaged as well.* (paragraph 7).

The behavior of term tARMED is specified in the *event table* below. Like mode transition tables, event tables make explicit only those events that cause the variable defined by the table to change. For example, the first entry in the first row states, "If mcStatus is ATTmode *or* FPAmode and mALTsw is turned on when tALTpresel is *true* and tNear is *false*, then tARMED becomes *true*." The entry "NEVER" in an event table means that no event can cause the variable defined by the table to assume the value in the same column as the entry; thus, the entry "NEVER" in row 2 of the table means that when mcStatus is ALTmode no event can cause tARMED to become *true*. An entry "@T(Inmode)" in a row of a mode transition table or an event table denotes the event "system entered the corresponding mode".

Modes	Events	
ATTmode, FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	@F(mcStatus = FPAmode)
ALTmode	NEVER	@F(mcStatus = FPAmode)
tARMED =	true	false

We finally present the behavior of the display cCASdisplay using the *condition table* below. This table states that "If tCASpresel is *true* then cCASdisplay has the value mCASdesired; otherwise, it has the value mCASactual". The complete autopilot specification is in Appendix B.

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCASactual

5 Discussion of General Issues

In [3] we present a verification technique for proving properties of SCR requirements specifications. This technique proved to be valuable in detecting and correcting bugs in the autopilot specification. For example, an initial formulation of the specification violated the property "*the altitude engage mode will be* ARMED *only when the flight path angle select mode is engaged*". The counterexample generated by the tool helped diagnose the error (we were setting tARMED to *true* when mcStatus is ALTmode, and mALTsw is turned on when tALTpresel is *true* and tNear is *false*).

We found that the PVS model does not clearly distinguish a system's environmental quantities from the dependent quantities. Also, by not clearly identifying environmental quantities the system monitors, and environmental quantities the system controls, it was very hard to find an answer to the question "What is the required behavior of the system?" by examining the PVS model. During the process of creating the SCR requirements specification, we came up with several questions for which we could not find answers from the PVS model. This is because the PVS description is not at the appropriate level of abstraction.

5.1 Appropriate Level of Abstraction

The PVS model of the autopilot in [4] is too *abstract* to serve as a requirements specification, i.e., as a black box description of all acceptable system implementations. Rather than specifying the required relationship between environmental quantities of the autopilot mode control panel, the PVS description is an abstract model of the mode control panel. Therefore, it is not a requirements specification. For example, the monitored quantity ALTactual is denoted abstractly by two boolean variables alt_reached and alt_gets_near; boolean variable input_alt abstractly denotes the pilot "dialing-in" the desired altitude using knob ALTdesired; etc. It is usual to make such abstractions during verification, because existing methods cannot be directly applied to requirements specifications, which are too detailed. However, the right approach is to begin by formulating the requirements specification, and later to describe formally the relationship between the specification and the abstract verification models. If the correspondence between the abstract models and the requirements specification is informal (or if the requirements specification is never created), it leaves room for misinterpretation.

5.2 Kinds of Analyses

In our experience, the first three phases of our idealized process for requirements analysis, viz., SRS Creation, SRS Checking, and SRS Validation, are the most crucial ones. It is very likely that a large proportion of activities of requirements analysis will be in support of these phases. It is also safe to assume that for a majority of projects (barring a small number of projects developing safety or mission critical applications) the last phase, i.e., SRS Verification, will be completely skipped. Since PVS concentrates exclusively on this phase of analysis, and provides poor support for the initial three phases, it is unlikely to be very effective as a tool to support requirements analysis. However, PVS has been effective in the analysis of critical algorithms and architectures for fault-tolerance, such as the correctness of distributed agreement protocols for a hybrid fault model, and in the verification of crucial subsystems, such as a commercial avionics microprocessor.

5.3 Role of Tool Support

In our experience, tools that support a limited analysis domain, with a specific conceptual model, tend to be more effective than general purpose tools. If a method lacks a strong underlying conceptual model, the benefits of automation are likely to be minimal ([8] provides more details). If a method does not adequately constrain the problem, the corresponding support tools cannot guide the developer when making difficult decisions. Since the SCR method standardizes the problem domain, the conceptual model, the notation, and the process, significant automated tool support is possible. For example, by using information about the current state of a specification, and knowledge of the process, a tool can guide developers in making the next step. Also, by providing standard templates, a tool can automate the routine activities of SRS creation. By applying the SCR method to several industrial problems, we plan to exploit the full potential of such tools.

6 Acknowledgements

We thank Jim Kirby for many helpful discussions on the autopilot specification. We gratefully acknowledge Ricky Butler for providing helpful insights and for his prompt answers to all our questions about the autopilot mode control panel.

References

[1] M. Alford. Software Requirements Engineering Methodology (Development). *RADC-TR-79-168*, U.S. Air Force Rome Air Development Center, June 1979.

[2] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software Requirements for the A-7E Aircraft. Technical Report NRL-9194, NRL, Washington DC, 1992.

[3] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. Submitted for publication.

[4] Ricky W. Butler. An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot. NASA Technical Memorandum 110255. NASA Langley Research Center, May 1996.

[5] B. L. DiVito and L. W. Roberts. Using Formal Methods to Assist in the Requirements Analysis of the Space Shuttle GPS Change Request. NASA Contractor Report 4752. NASA Langley Research Center, Hampton VA 23681, August 1996.

[6] S. R. Faulk, et al. The CoRE method for real-time requirements. *IEEE Software*, 9(5), September 1992.

[7] S. R. Faulk, et al. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conference on Computer Assurance,* Gaithersburg MD, June 1994.

[8] S. R. Faulk. Software Requirements: A Tutorial. Technical Report *NRL/MR/5546-95-7775*, Naval Research Laboratory, Washington DC, 1995.

[9] General Accounting Office (US). Mission Critical Systems: Defense Attempting to Address Major Software Challenges. *GAO/IMTEC-93-12*, December 1992.

[10] Constance Heitmeyer, et al. SCR*: A toolset for specifying and analyzing requirements. In *Proc.* 10th *Annual Conference on Computer Assurance*, NIST, Gaithersburg MD, June 1995.

[11] K. L. Heninger. "Specifying software requirements for complex systems: New techniques and their applications". *IEEE Transactions on Software Engineering* SE-6(1), Jan 1980.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for Analyzing SCR-style Requirements Specifications: A Formal Foundation. Technical Report NRL-7499, NRL, Wash. DC, 1995. In preparation.

[13] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM Trans. on Software Engg. and Methodology*, 5(3)231–261, July 1996.

[14] Constance Heitmeyer, Bruce Labaw, d Daniel Kiskis. Consistency checking of SCR-style requirements specifications. In *Proc. 1995 Int'l Symposium on Requirements Engg.*, York, England, March 1995.

[15] C. L. Heitmeyer and J. McLean. "Abstract requirements specifications: A new approach and its application". *IEEE Transactions on Software Engineering*, SE-9(5), Sep 1983.

[16] R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *Proc. IEEE Int'l Symp. on Requirements Engg.*, pp. 126–133, Jan 1993.

[17] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In 11th *International Conference on Automated Deduction*, LNCS-607, pp 748–752, 1992.

[18] D. L. Parnas, G. J. K. Asmis and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety,* 32(2), 1991.

[19] D. L. Parnas and P. Clements. A rational design process: how and why to fake it. *IEEE Trans. on Software Engg.*, 12(2), February 1986.

[20] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming,* 25(1), pp 41–62, Oct 1995.

[21] M. K. Srivas and S. P. Miller. Formal Verification of an Avionics Microprocessor. NASA Contractor Report 4682, NASA Langley Research Center, July 1995.

[22] W. D. Young. Comparing verification systems: interactive consistency in ACL2. In *Proc. COMPASS'96*, Gaithersburg MD, 1996.

A Description of the autopilot

1. *The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values, as shown in Figure 1. The system supports the following four modes: attitude control wheel steering (*`ATTmode`*), flight path angle selected (*`FPAmode`*), altitude engage (*`ALTmode`*), and calibrated air speed (*`CASmode`*).*

   Only one of the first three modes can be engaged at any time. The mode `CASmode` *can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes,* `ATTmode`, `FPAmode`, *or* `ALTmode` *should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.*

2. *There are three displays on the panel: altitude (*`ALTdisplay`*), flight path angle (*`FPAdisplay`*), and calibrated air speed (*`CASdisplay`*). The displays usually show the current values of altitude (*`ALTactual`*), flight path angle (*`FPAactual`*), and air speed (*`CASactual`*) of the aircraft. However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display (*`ALTdesired`, `FPAdesired`, *or* `CASdesired`*). This is the target or "pre-selected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to* 25,000 *feet, he will dial* 25,000 *(using the knob* `ALTdesired`*) into* `ALTdisplay` *and then press* `ALTsw` *to engage* `ALTmode`*. Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value.*

3. *If the pilot dials into* `ALTdesired` *an altitude that is more than* 1,200 *feet above the current altitude (*`ALTactual`*) and then presses* `ALTsw`*, then* `ALTmode` *will not directly engage. Instead, the altitude engage mode will change to "armed" and* `FPAmode` *is engaged. The pilot must then dial in, using the knob* `FPAdesired`*, the desired flight-path angle into* `FPAdisplay`*, which will be followed by the flight-control system until the aircraft attains the desired altitude.* `FPAmode` *will remain engaged until the aircraft is within* 1,200 *feet of* `ALTactual`*, then* `ALTmode` *is automatically engaged.*

4. `CASdesired` *and* `FPAdesired` *need not be pre-selected before the corresponding modes are engaged — the current values displayed will be used. The pilot can dial-in a different target value after the mode is engaged. However, the altitude must be pre-selected before* `ALTsw` *is pressed. Otherwise, the command is ignored.*

5. `CASsw` *and* `FPAsw` *toggle on and off every time they are pressed. For example, if* `CASsw` *is pressed while the system is already in* `CASmode`*, that mode will be disengaged. However, if* `ATTsw` *is pressed while* `ATTmode` *is already engaged, the command is ignored. Likewise, pressing* `ALTsw` *while the system is already in* `ALTmode` *has no effect.*

6. *Whenever a mode other than* `CASmode` *is engaged, all other pre-selected displays should return to current.*

7. *If the pilot dials in a new altitude while* `ALTmode` *is engaged or the altitude engage mode is "armed", then* `ALTmode` *is disengaged and* `ATTmode` *is engaged. If the altitude engage mode is "armed" then* `FPAmode` *should be disengaged as well.*

B SCR Specfication of the autopilot

Monitored Variables:

mALTactual, mCASactual, mFPAactual : Integer **initially** all 0;
mALTsw, mATTsw, mCASsw, mFPAsw : {on, off} **initially** all off;
mALTdesired, mCASdesired, mFPAdesired : Integer **initially** all 0;

Controlled Variables:

cALTdisplay, cCASdisplay, cFPAdisplay : Integer **initially** all 0;

Mode Class:

mcStatus : {ALTmode, ATTmode, FPAmode} **initially** ATTmode;

Terms:

tARMED : Boolean **initially** false;
tCASmode : Boolean **initially** false;
tALTpresel, tCASpresel, tFPApresel : Boolean **initially** all false;
tNear $\overset{\text{def}}{=}$ mALTdesired $-$ mALTactual ≤ 1200;

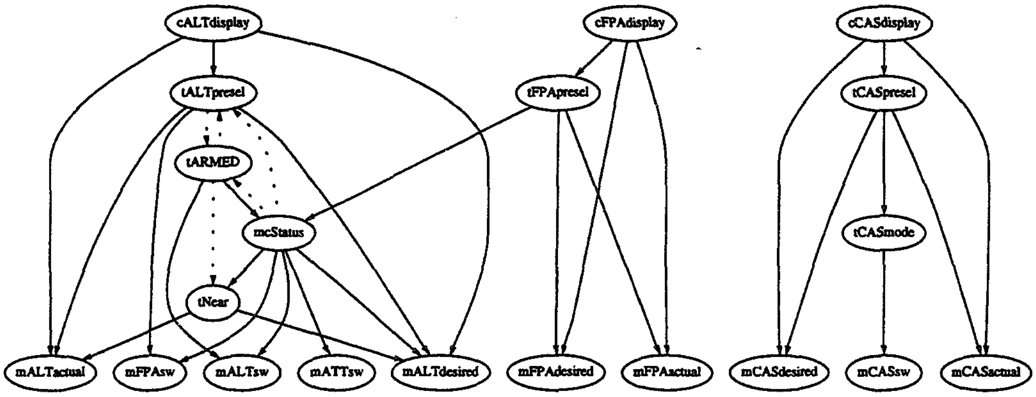


Figure 2: Variable Dependency Graph

Mode Transition Table for mcStatus		
Source Mode	Events	Destination Mode
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode
ALTmode	@T(mFPAsw = on)	FPAmode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
ATTmode	@T(mFPAsw = on) OR @T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAmode
FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear) OR @T(tNear) WHEN tARMED	ALTmode
FPAmode	@T(mATTsw = on) OR @T(mFPAsw = on) OR CHANGED(mALTdesired) WHEN tARMED	ATTmode

Modes	Events	
ATTmode, FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	@F(mcStatus = FPAmode)
ALTmode	NEVER	@F(mcStatus = FPAmode)
tARMED =	true	false

	Events	
	@T(mCASsw = on) WHEN NOT tCASmode	@T(mCASsw = on) WHEN tCASmode
tCASmode =	true	false

Modes	Events	
ALTmode	NEVER	@T(mALTdesired = mALTactual) OR @F(INMODE)
FPAmode	CHANGED(mALTdesired) WHEN NOT tARMED	NEVER
ATTmode	CHANGED(mALTdesired)	@T(INMODE) OR @T(mFPAsw = on)
tALTpresel =	true	false

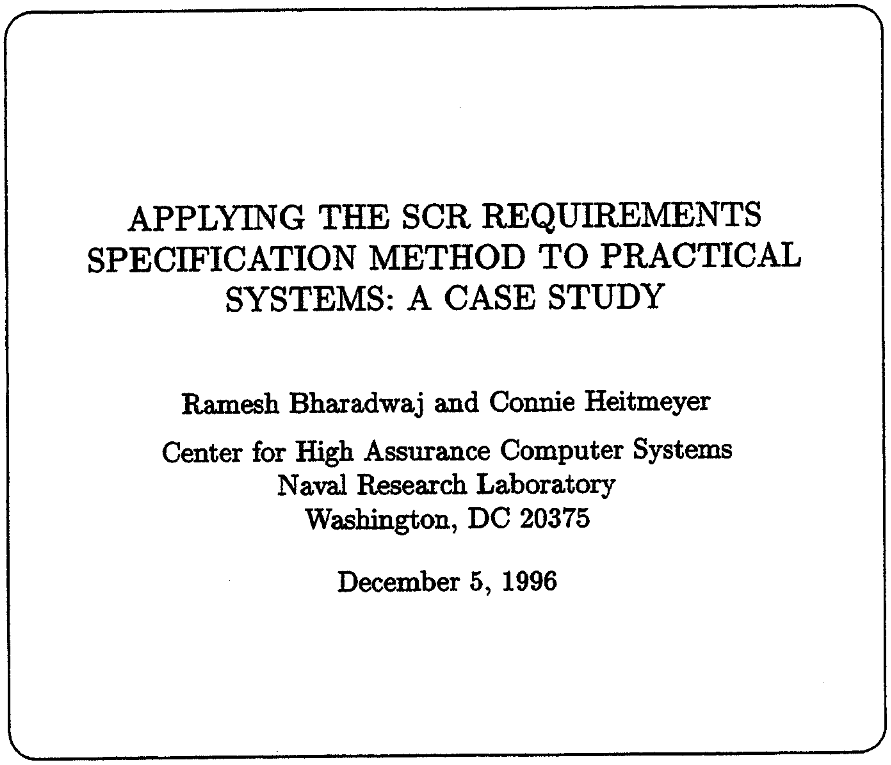
	Events	
	CHANGED(mCASdesired)	@F(tCASmode) OR @T(mCASdesired = mCASactual) WHEN tCASmode
tCASpresel =	true	false

	Events	
	CHANGED(mFPAdesired)	@T(mcStatus = ATTmode) OR @T(mcStatus = ALTmode) OR @T(mFPAdesired = mFPAactual) WHEN (mcStatus = FPAmode)
tFPApresel =	true	false

	Conditions	
	tALTpresel	NOT tALTpresel
cALTdisplay =	mALTdesired	mALTactual

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCASactual

	Conditions	
	tFPApresel	NOT tFPApresel
cFPAdisplay =	mFPAdesired	mFPAactual

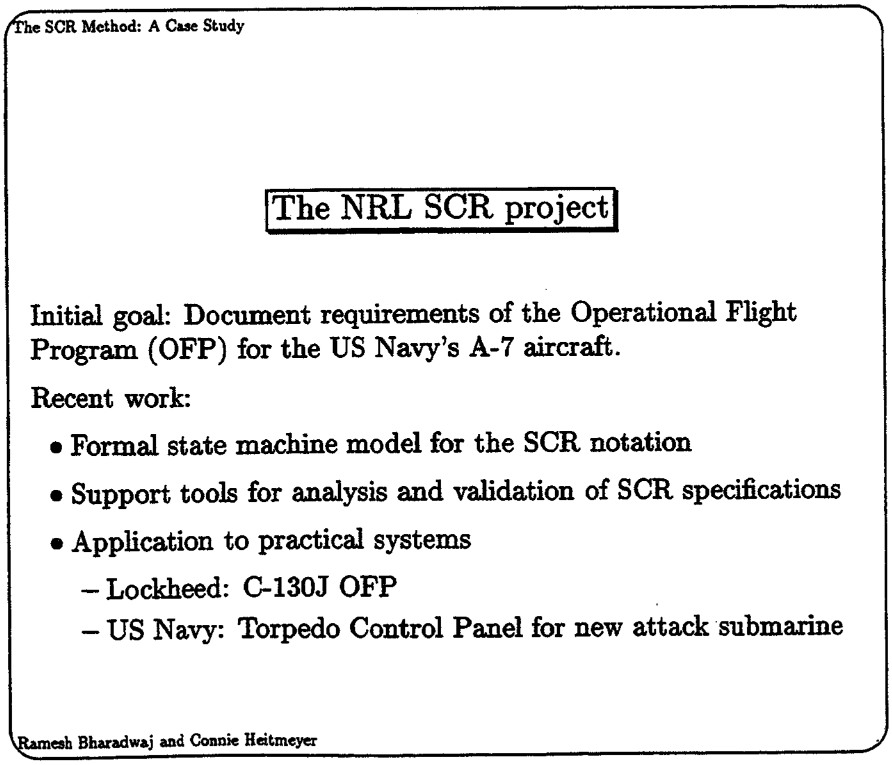


APPLYING THE SCR REQUIREMENTS SPECIFICATION METHOD TO PRACTICAL SYSTEMS: A CASE STUDY

Ramesh Bharadwaj and Connie Heitmeyer

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375

December 5, 1996

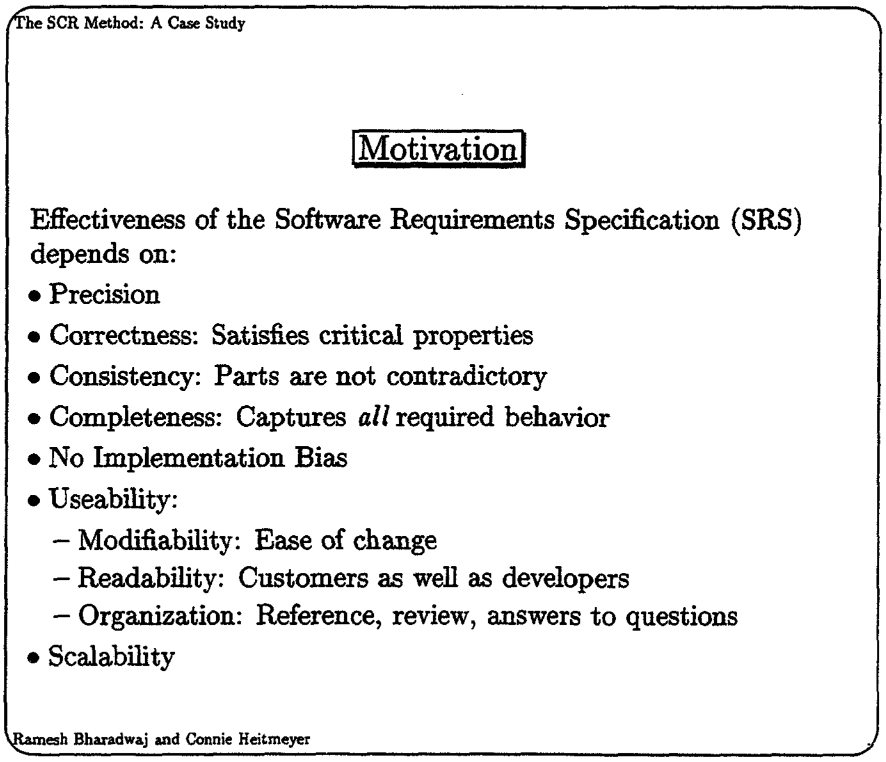


The NRL SCR project

Initial goal: Document requirements of the Operational Flight Program (OFP) for the US Navy's A-7 aircraft.

Recent work:

- Formal state machine model for the SCR notation
- Support tools for analysis and validation of SCR specifications
- Application to practical systems
  - Lockheed: C-130J OFP
  - US Navy: Torpedo Control Panel for new attack submarine



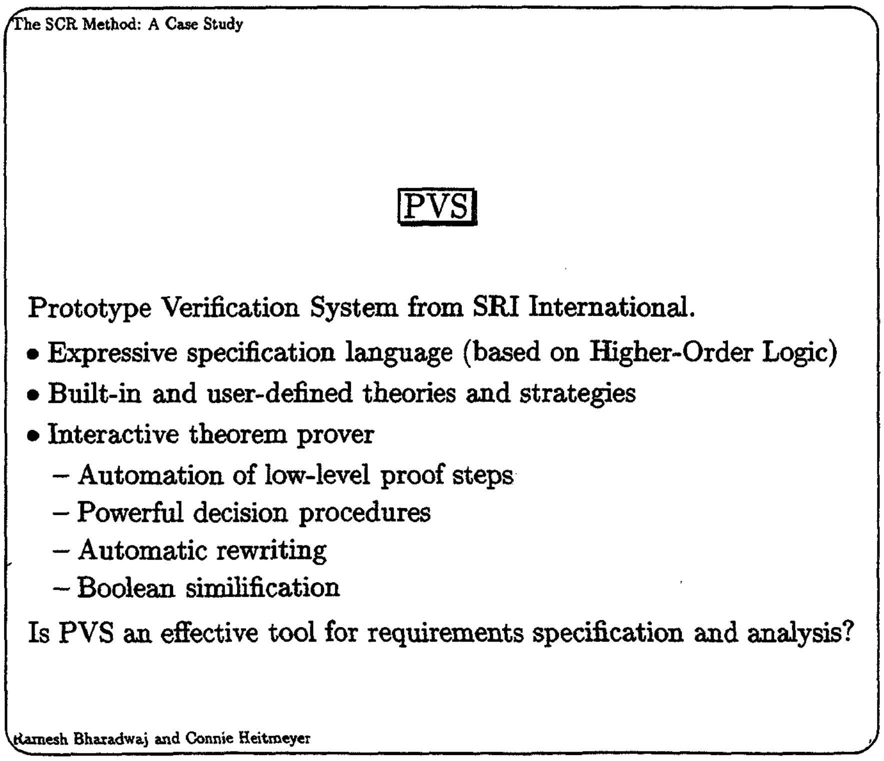
The SCR Method: A Case Study

Motivation

Effectiveness of the Software Requirements Specification (SRS) depends on:

- Precision
- Correctness: Satisfies critical properties
- Consistency: Parts are not contradictory
- Completeness: Captures *all* required behavior
- No Implementation Bias
- Useability:
  - Modifiability: Ease of change
  - Readability: Customers as well as developers
  - Organization: Reference, review, answers to questions
- Scalability

Ramesh Bharadwaj and Connie Heitmeyer



The SCR Method: A Case Study

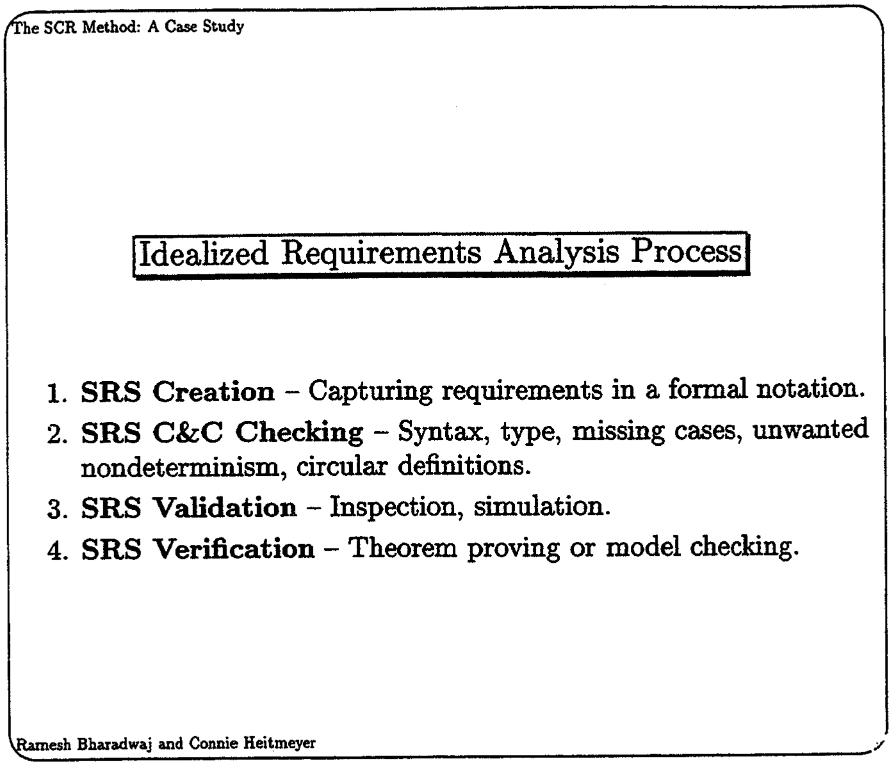
PVS

Prototype Verification System from SRI International.

- Expressive specification language (based on Higher-Order Logic)
- Built-in and user-defined theories and strategies
- Interactive theorem prover
  - Automation of low-level proof steps
  - Powerful decision procedures
  - Automatic rewriting
  - Boolean similification

Is PVS an effective tool for requirements specification and analysis?

Ramesh Bharadwaj and Connie Heitmeyer

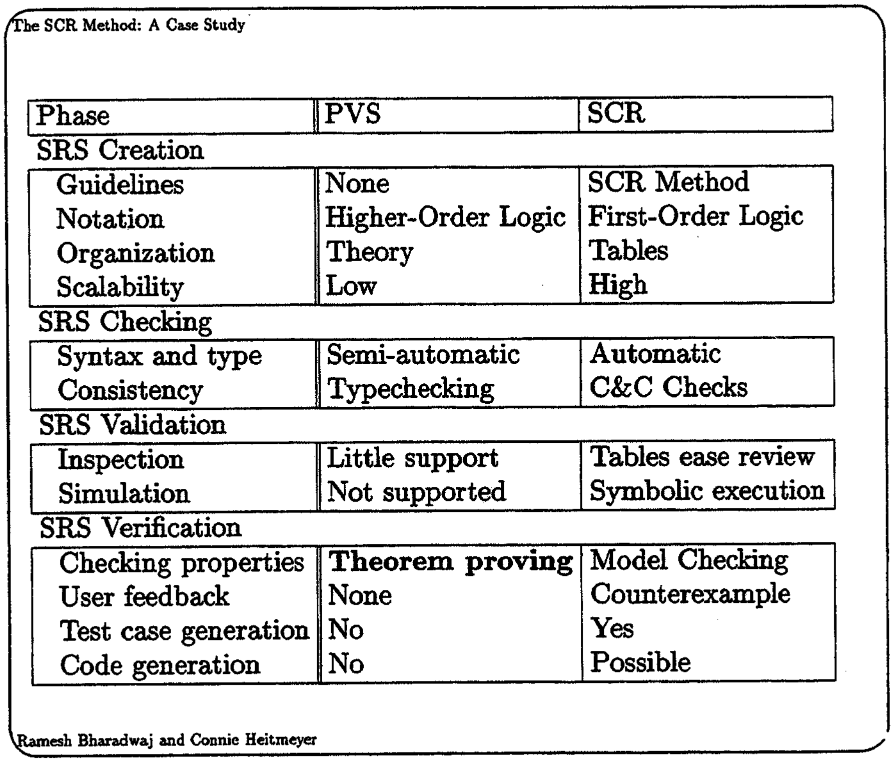


The SCR Method: A Case Study

Idealized Requirements Analysis Process

1. **SRS Creation** – Capturing requirements in a formal notation.
2. **SRS C&C Checking** – Syntax, type, missing cases, unwanted nondeterminism, circular definitions.
3. **SRS Validation** – Inspection, simulation.
4. **SRS Verification** – Theorem proving or model checking.

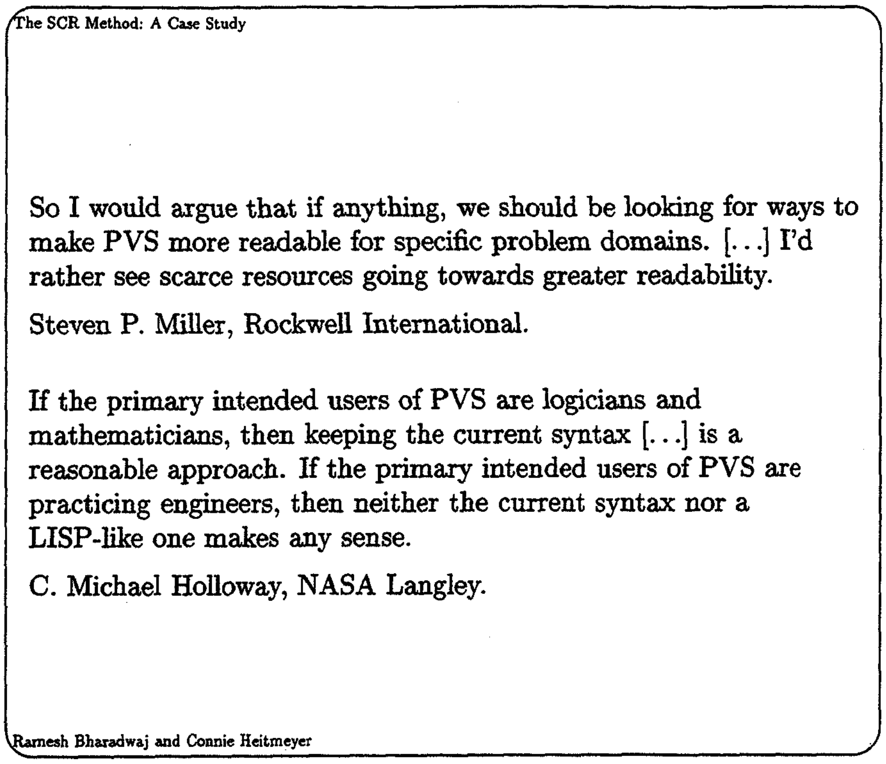
Ramesh Bharadwaj and Connie Heitmeyer



The SCR Method: A Case Study

Phase	PVS	SCR
SRS Creation		
Guidelines	None	SCR Method
Notation	Higher-Order Logic	First-Order Logic
Organization	Theory	Tables
Scalability	Low	High
SRS Checking		
Syntax and type	Semi-automatic	Automatic
Consistency	Typechecking	C&C Checks
SRS Validation		
Inspection	Little support	Tables ease review
Simulation	Not supported	Symbolic execution
SRS Verification		
Checking properties	**Theorem proving**	Model Checking
User feedback	None	Counterexample
Test case generation	No	Yes
Code generation	No	Possible

Ramesh Bharadwaj and Connie Heitmeyer



The SCR Method: A Case Study

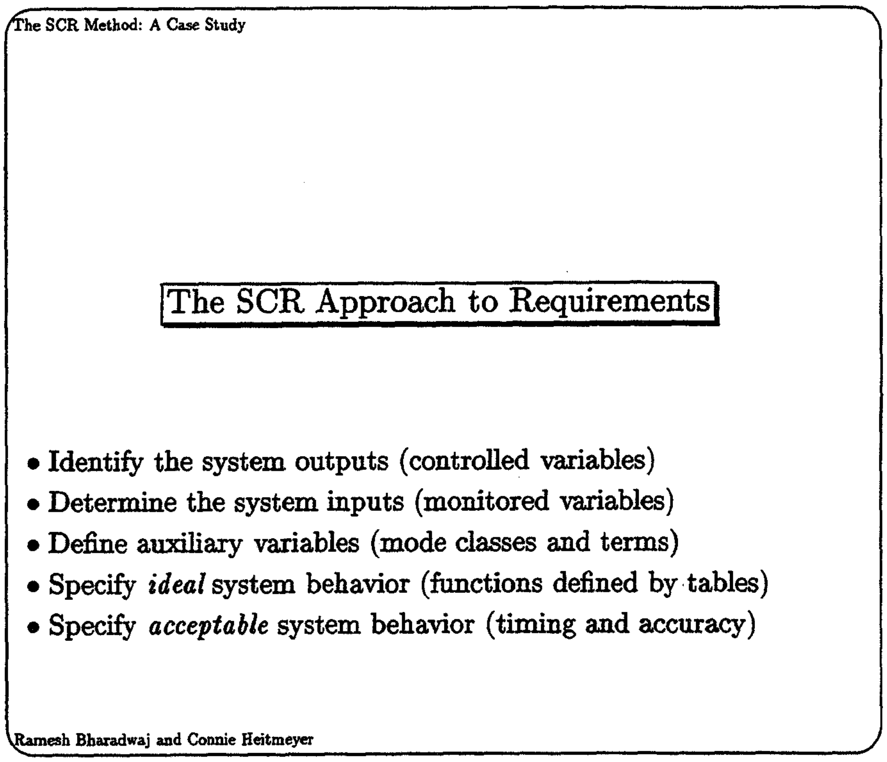
So I would argue that if anything, we should be looking for ways to make PVS more readable for specific problem domains. [...] I'd rather see scarce resources going towards greater readability.

Steven P. Miller, Rockwell International.

If the primary intended users of PVS are logicians and mathematicians, then keeping the current syntax [...] is a reasonable approach. If the primary intended users of PVS are practicing engineers, then neither the current syntax nor a LISP-like one makes any sense.

C. Michael Holloway, NASA Langley.

Ramesh Bharadwaj and Connie Heitmeyer



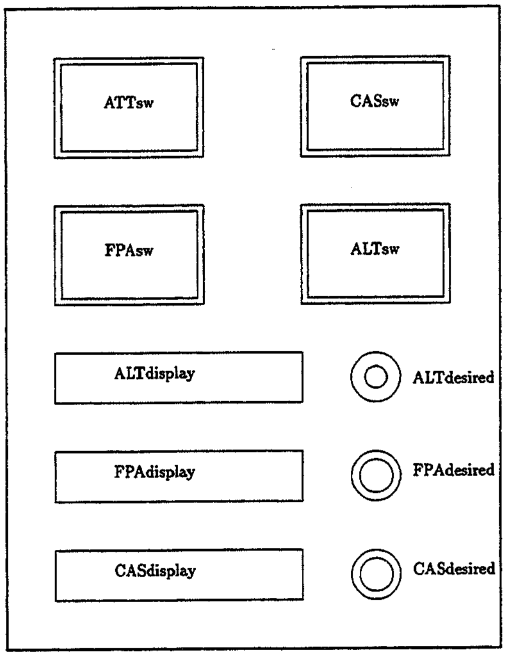
The SCR Method: A Case Study

The SCR Approach to Requirements

- Identify the system outputs (controlled variables)
- Determine the system inputs (monitored variables)
- Define auxiliary variables (mode classes and terms)
- Specify *ideal* system behavior (functions defined by tables)
- Specify *acceptable* system behavior (timing and accuracy)

Ramesh Bharadwaj and Connie Heitmeyer

The SCR Method: A Case Study



Ramesh Bharadwaj and Connie Heitmeyer

The SCR Method: A Case Study

Monitored Variables

mALTactual, **mCASactual**, **mFPAactual** : Integer;
mALTsw, **mATTsw**, **mCASsw**, **mFPAsw** : $\{on, off\}$;
mALTdesired, **mCASdesired**, **mFPAdesired** : Integer;

Controlled Variables

cALTdisplay, **cCASdisplay**, **cFPAdisplay** : Integer;

Mode Class

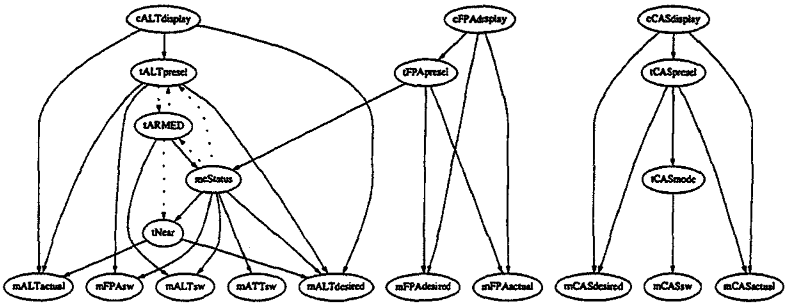
mcStatus : $\{ALTmode, ATTmode, FPAmode\}$;

Terms

tArmed : Boolean;
tCASmode : Boolean;
tALTpresel, **tCASpresel**, **tFPApresel** : Boolean;
tNear : Boolean;

Ramesh Bharadwaj and Connie Heitmeyer

The SCR Method: A Case Study



Ramesh Bharadwaj and Connie Heitmeyer

The SCR Method: A Case Study

Mode Transition Table for mcStatus		
Source Mode	Events	Dest. Mode
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode
ALTmode	@T(mFPAsw = on)	FPAmode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
ATTmode	@T(mFPAsw = on) OR @T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAmode
FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear) OR @T(tNear) WHEN tARMED	ALTmode
FPAmode	@T(mATTsw = on) OR @T(mFPAsw = on) OR CHANGED(mALTdesired) WHEN tARMED	ATTmode

Ramesh Bharadwaj and Connie Heitmeyer

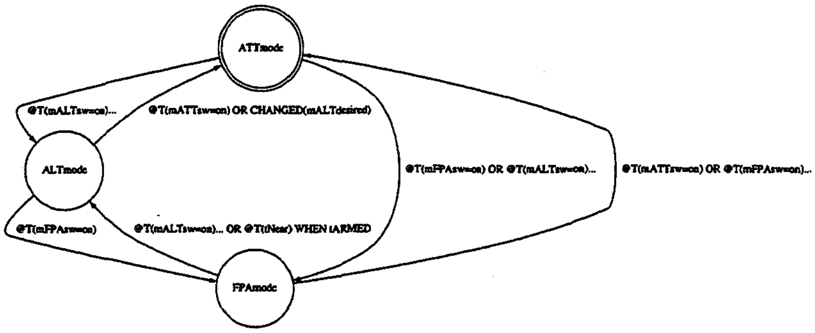
The SCR Method: A Case Study

Mode Transition Table for mcStatus		
Source Mode	Events	Destination Mode
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode

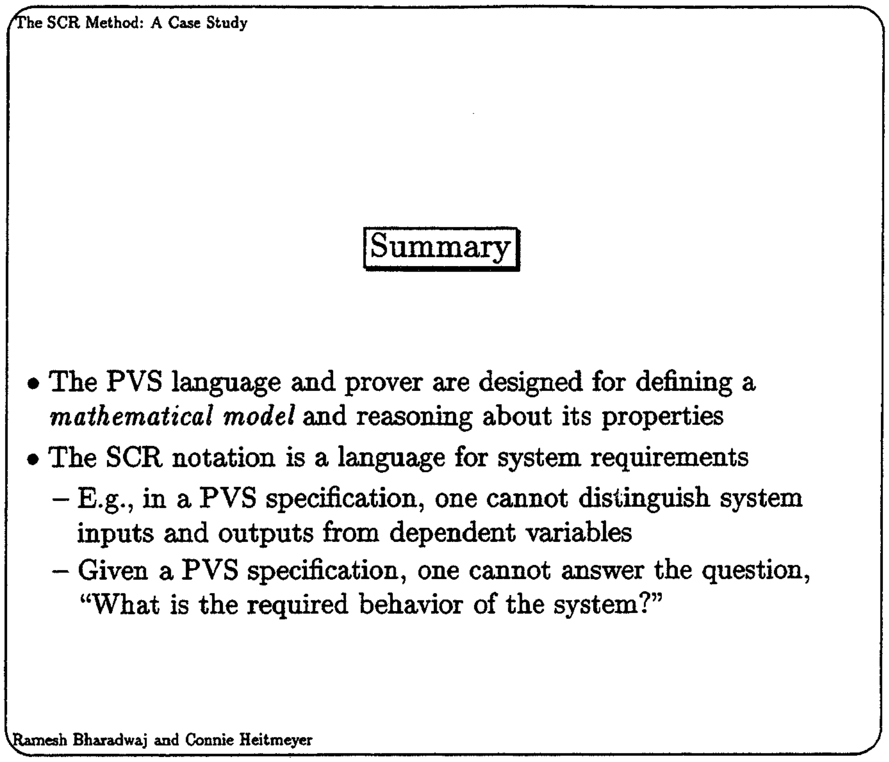
- *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., pressing ATTsw should engage ATTmode **OR** *If the pilot dials in a new altitude while* ALTmode *is engaged, then* ALTmode *is disengaged and* ATTmode *is engaged* (paragraph 7).

Ramesh Bharadwaj and Connie Heitmeyer

The SCR Method: A Case Study


ATTmode
@T(mALTsw==on)...
@T(mATTsw==on) OR CHANGED(mALTdesired)
ALTmode
@T(mFPAsw==on) OR @T(mALTsw==on)...
@T(mATTsw==on) OR @T(mFPAsw==on)...
@T(mFPAsw==on)
@T(mALTsw==on)... OR @T(tNear) WHEN tARMED
FPAmode

Ramesh Bharadwaj and Connie Heitmeyer



Summary

- The PVS language and prover are designed for defining a *mathematical model* and reasoning about its properties
- The SCR notation is a language for system requirements
  - E.g., in a PVS specification, one cannot distinguish system inputs and outputs from dependent variables
  - Given a PVS specification, one cannot answer the question, "What is the required behavior of the system?"

Panel Discussion: Transferring Best Practices: Why Is It So Complex?

omit to end

Moderator: Vic Basili, University of Maryland

Richard DeMillo, Bellcore

Michael Evangelist, Florida International University

Peter Freeman, Georgia Institute of Technology

Allan Willey, Motorola Corporation

SEL 21 Panel
Transferring Best Practices: Why is it so hard?

Panelists:

Richard DeMillo	Bellcore
Michael Evangelist	Florida International University
Peter Freeman	Georgia Tech
Allan Willey	Motorola

Premise:

Transferring any technology is very hard. In fact it has been harder than most people and organizations believe. For this reason, many organizations are unwilling to admit how unsuccessful they have been in transferring or sustaining best practices. We would like the panel to react to this premise.

SEL 21 Panel
Transferring Best Practices: Why is it so hard?

Each Panelist was asked to:

Give your background and experience with technology transfer.

Give one or two specific examples of transfer projects you have observed or participated in:
what procedures were followed in the transfer,
what organizations were involved,
what were the major problems,
what was the cost in time and schedule,
what were the results,
what was the reaction of the participants,
what aspects can you demonstrate was successful,
what would you do differently now?

What can you share with the audience in terms of lessons learned?

SEL Panel

5 December 1996
NASA/Goddard
Richard DeMillo, Bellcore

Contents

- Bellcore background
- Examples of Successful Transfer
  - Adapt/XAdvertiser
  - xATAC
  - Programmability
- Web speed and change
- Carddiagram
- Team vs Transfer
- Adapt/XModel

Bellcore background

- Divestiture and Sale
- Core customers and new customers
- Product Lines and Development Processes
- ISO and CMM

Successful Transfer: Advertiser

- Describe and Market
- SS/AR team formation
- AR led with business case
  - advertising would be key
  - scaleability
  - right commercial model

Successful Transfer: xATAC

- AR advocate--top to bottom
- Did not start from coverage--backing into it
- Team formation--fear as the motivator
- AR led with business case
  - – third party validation
  - – cost-benefit
  - – scaleability

Successful Transfer: Programmability

- Long-term research on declarative optimization
- Competitive opportunity
- LAURE was on the shelf
- Scaleable technology

Web Speed and Change

- 18 mos to 4 mos development cycles
- 75% solutions that can evolve quickly
- Platforms and features
- Version 1.0 is part of requirements definition
- Ascendancy of architecture (eg availability, scaleability
- RAD and Card diagram

Team vs. Transfer

- No time for transfer
- Investment in inventory
- Radar Screens and accountability
- Impact of RAD
- Adapt/XModel

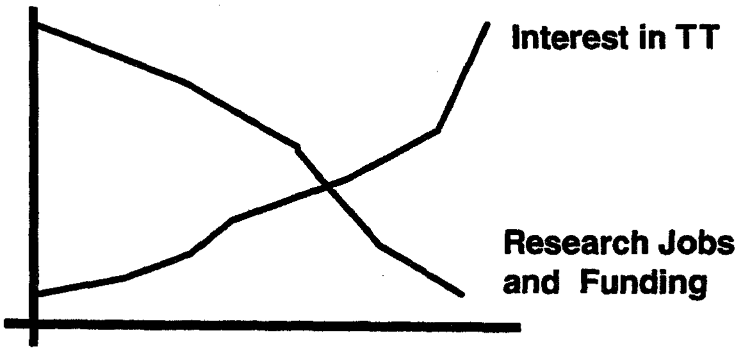
Product vs. Process

- P&L managers are easier to influence
- Institutional change not needed
- Adapt/XModel is a result of transfer

Transferring Research Technology

Michael Evangelist
School of Computer Science
Florida International University
Miami, FLA

Personal Observation



When Do I Transfer Technology?

- Need to believe that the new technology
  - solves a problem
  - works
  - fits
  - has low "cost"

Three Transfer Examples

- VERDI, 1985-90 (research prototype)
- BAL/SRW, 1990-92 (advanced development)
- PC networking application, December 1996 (product)

VERDI

- Graphical tool for designing distributed systems
  - does all the right things, simply
- We did it the right way
- <u>Result</u>: lots of interest, no serious use
  - not commercial quality
  - solved part of the problem
  - didn't fit environment or culture
  - platform and training costs high

BAL/SRW

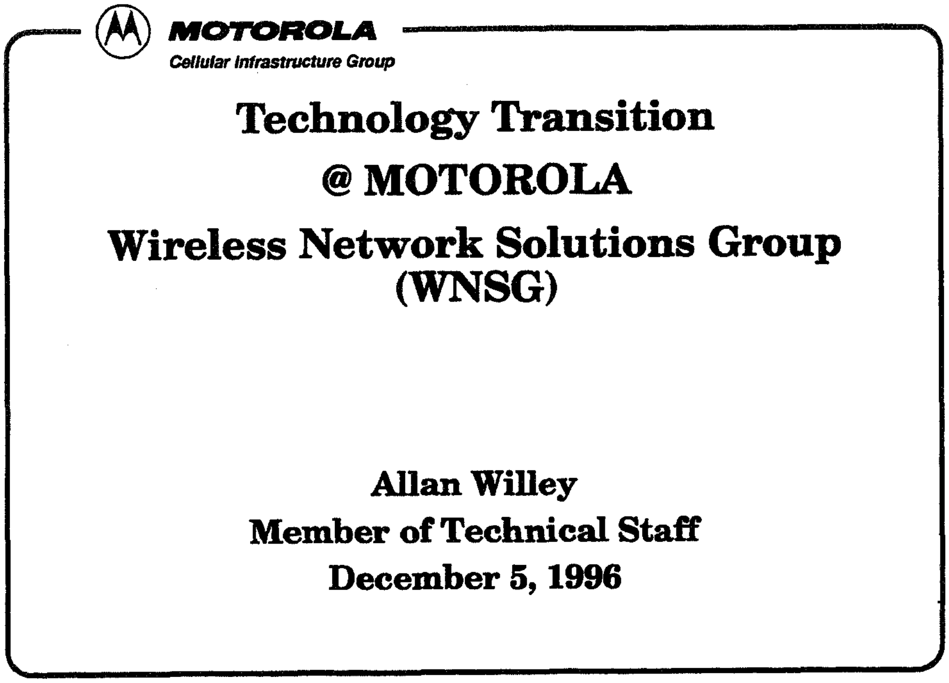
- Workbench for re-engineering legacy BAL programs
  - useful, graphical, status quo
- We worked closely with users
- <u>Result</u>: substantial use at a few clients
  - much closer to product level
  - solved urgent problem
  - fit culture but not computing environment
  - hardware cost high

PC Networking Application

- Establishes PPP connection over phone line
- Numerous hard-to-find bugs, poor technical help
- Result: no limit on the amount of effort I'll put into it
  - doesn't work, but we're optimistic
  - solves important problem
  - fits system and culture
  - low long-term cost

Observations

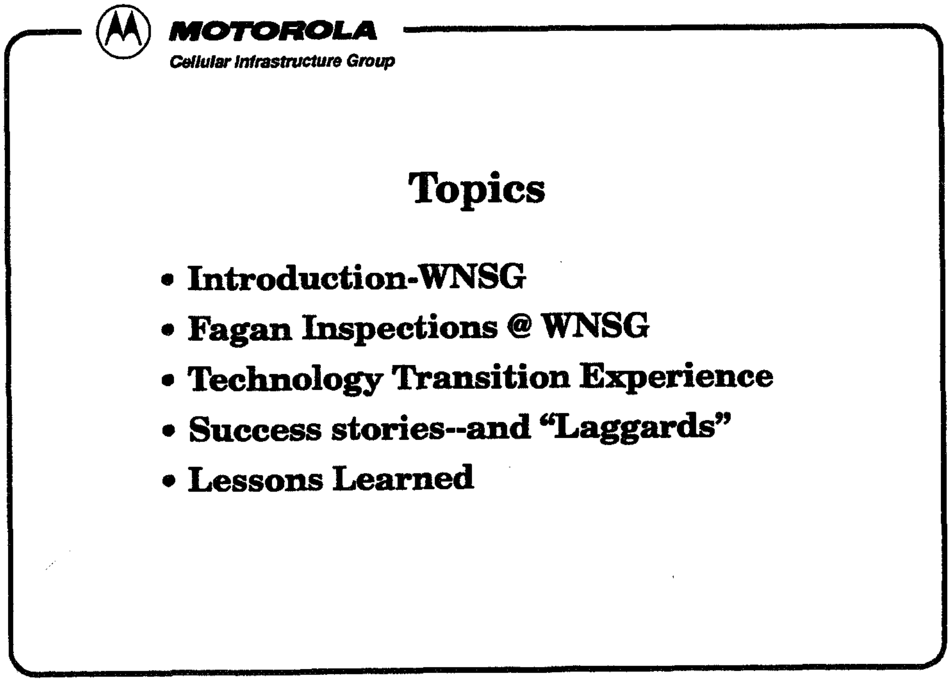
- Motivation of researchers now less of a problem
- Education of software engineers a serious concern
- TT model
  - working engineers educated in standard practice
  - research preps engineering years in advance
  - if you're not inventing sliced bread, resign yourself to incremental transfers



MOTOROLA
Cellular Infrastructure Group

Technology Transition
@ MOTOROLA
Wireless Network Solutions Group (WNSG)

Allan Willey
Member of Technical Staff
December 5, 1996

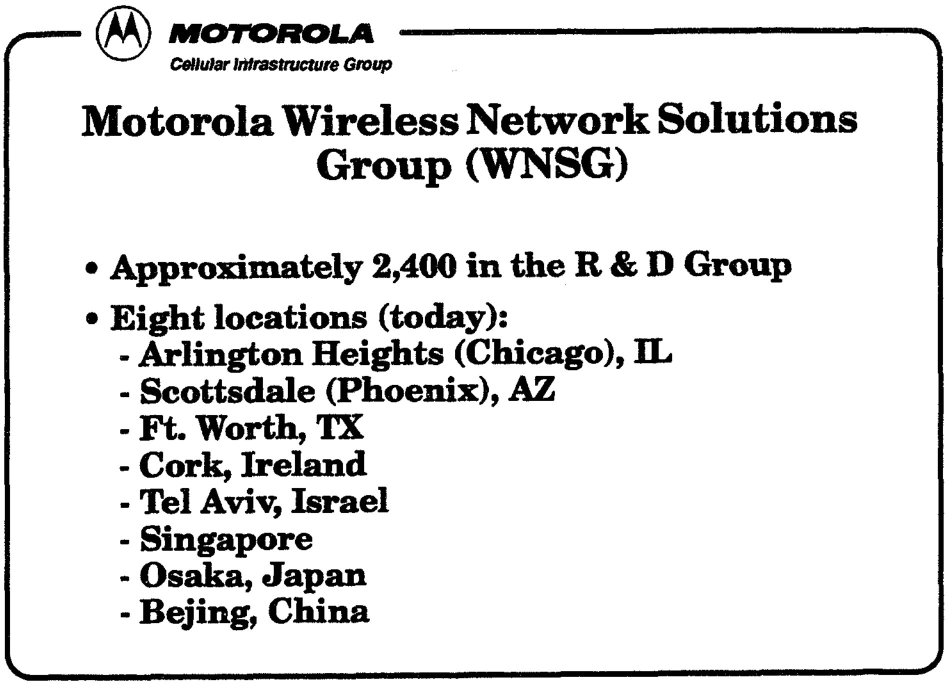


MOTOROLA
Cellular Infrastructure Group

Topics

- Introduction-WNSG
- Fagan Inspections @ WNSG
- Technology Transition Experience
- Success stories--and "Laggards"
- Lessons Learned

21st SEW Panel--12/3/96

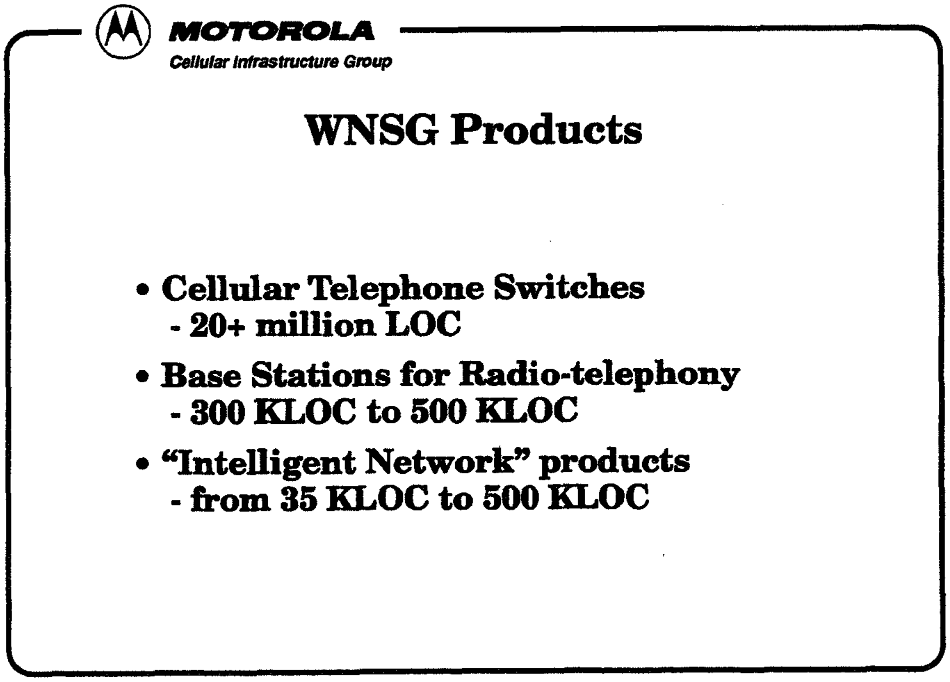


MOTOROLA
Cellular Infrastructure Group

Motorola Wireless Network Solutions Group (WNSG)

- Approximately 2,400 in the R & D Group
- Eight locations (today):
  - Arlington Heights (Chicago), IL
  - Scottsdale (Phoenix), AZ
  - Ft. Worth, TX
  - Cork, Ireland
  - Tel Aviv, Israel
  - Singapore
  - Osaka, Japan
  - Bejing, China

21st SEW Panel--12/3/96

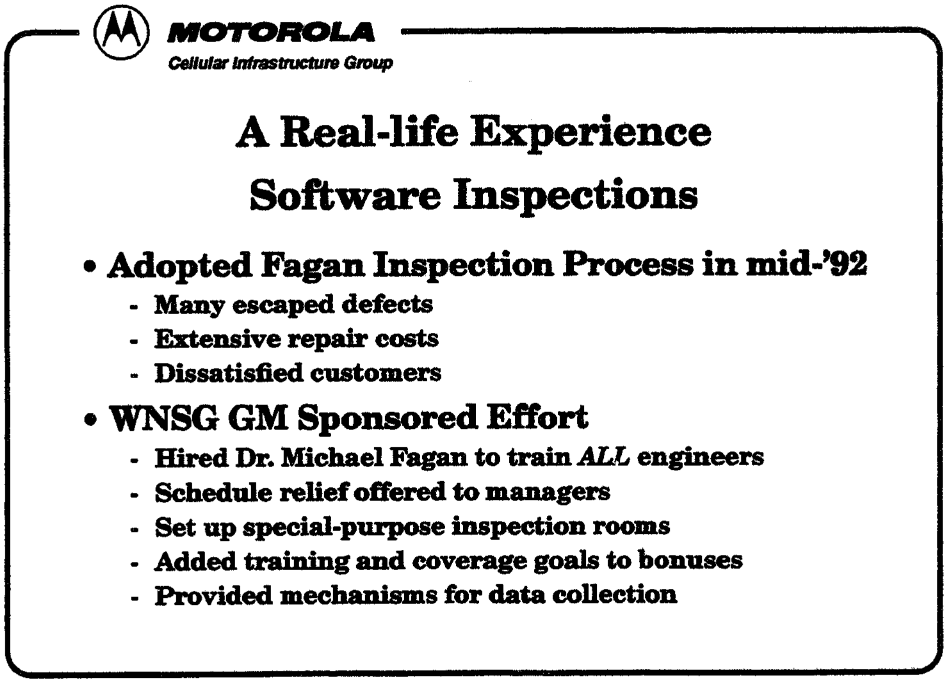


MOTOROLA
Cellular Infrastructure Group

WNSG Products

- Cellular Telephone Switches
  - 20+ million LOC
- Base Stations for Radio-telephony
  - 300 KLOC to 500 KLOC
- "Intelligent Network" products
  - from 35 KLOC to 500 KLOC

21st SEW Panel--12/3/96

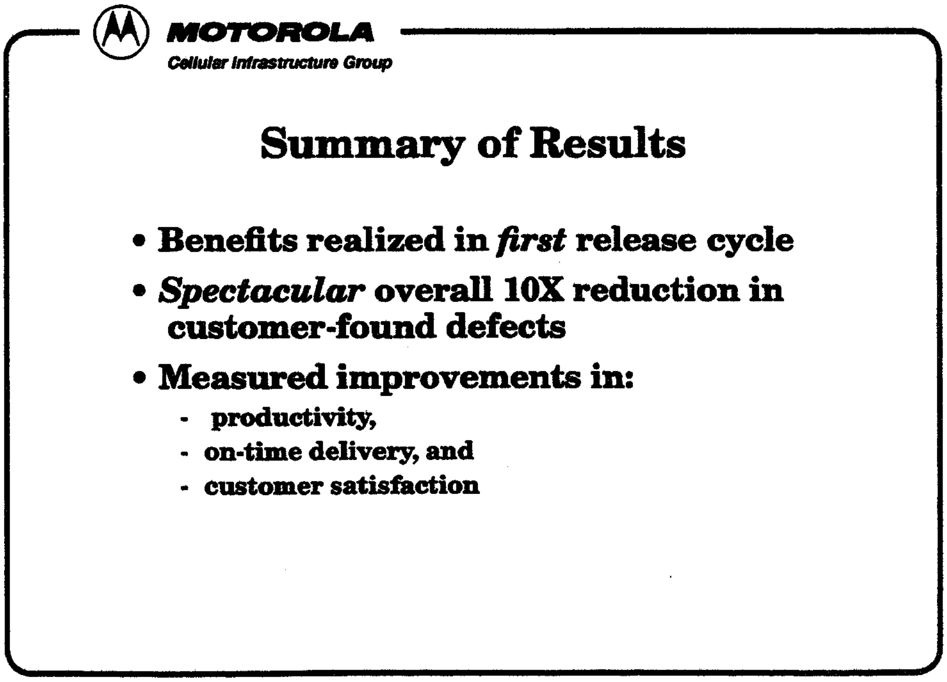


MOTOROLA
Cellular Infrastructure Group

A Real-life Experience Software Inspections

- **Adopted Fagan Inspection Process in mid-'92**
  - Many escaped defects
  - Extensive repair costs
  - Dissatisfied customers
- **WNSG GM Sponsored Effort**
  - Hired Dr. Michael Fagan to train *ALL* engineers
  - Schedule relief offered to managers
  - Set up special-purpose inspection rooms
  - Added training and coverage goals to bonuses
  - Provided mechanisms for data collection

21st SEW Panel--12/3/96

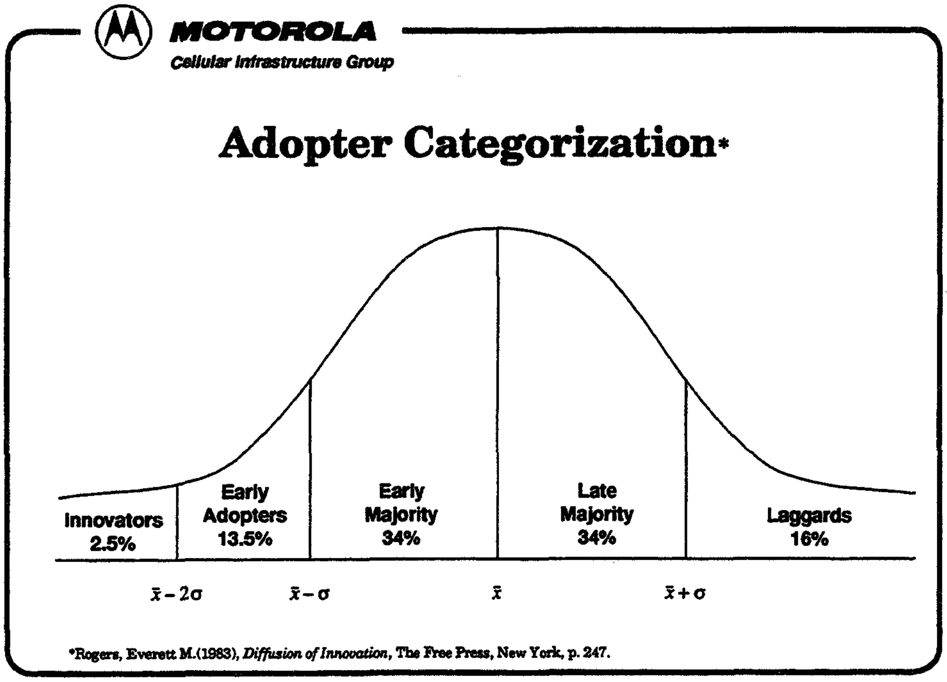


MOTOROLA
Cellular Infrastructure Group

Summary of Results

- **Benefits realized in *first* release cycle**
- ***Spectacular* overall 10X reduction in customer-found defects**
- **Measured improvements in:**
  - productivity,
  - on-time delivery, and
  - customer satisfaction

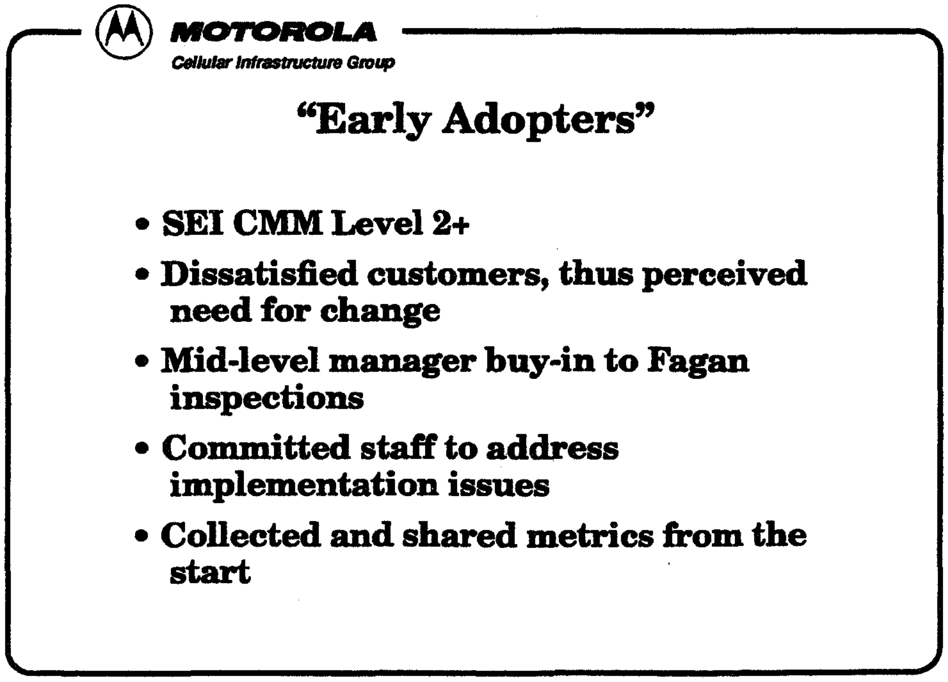
21st SEW Panel--12/3/96


MOTOROLA
Cellular Infrastructure Group

Adopter Categorization*

*Rogers, Everett M.(1983), *Diffusion of Innovation*, The Free Press, New York, p. 247.

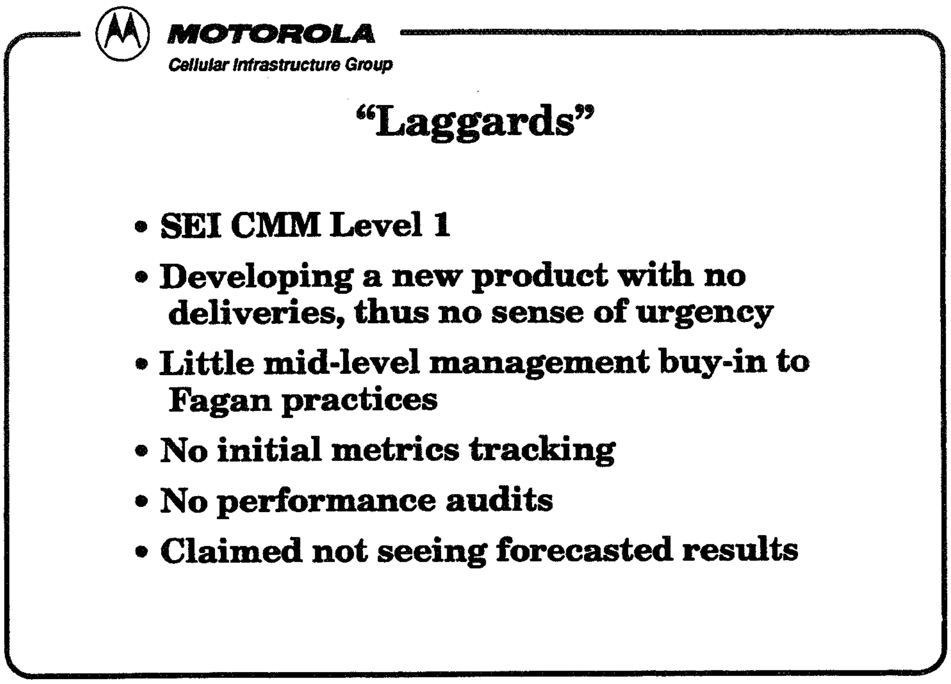
21st SEW Panel--12/3/96


MOTOROLA
Cellular Infrastructure Group

"Early Adopters"

- **SEI CMM Level 2+**
- **Dissatisfied customers, thus perceived need for change**
- **Mid-level manager buy-in to Fagan inspections**
- **Committed staff to address implementation issues**
- **Collected and shared metrics from the start**

21st SEW Panel--12/3/96

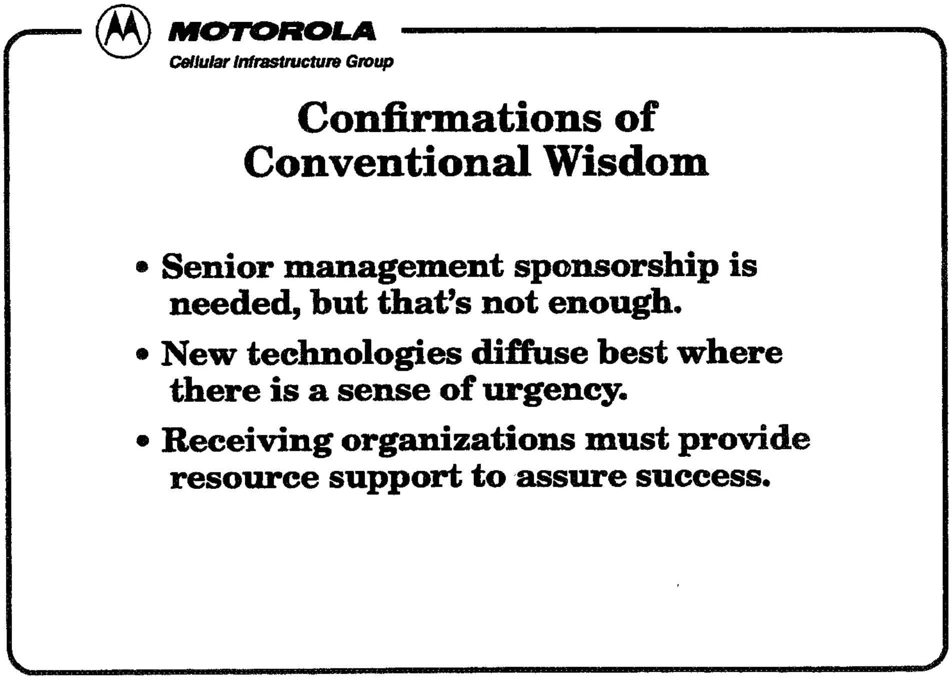


MOTOROLA
Cellular Infrastructure Group

"Laggards"

- SEI CMM Level 1
- Developing a new product with no deliveries, thus no sense of urgency
- Little mid-level management buy-in to Fagan practices
- No initial metrics tracking
- No performance audits
- Claimed not seeing forecasted results

21st SEW Panel--12/3/96

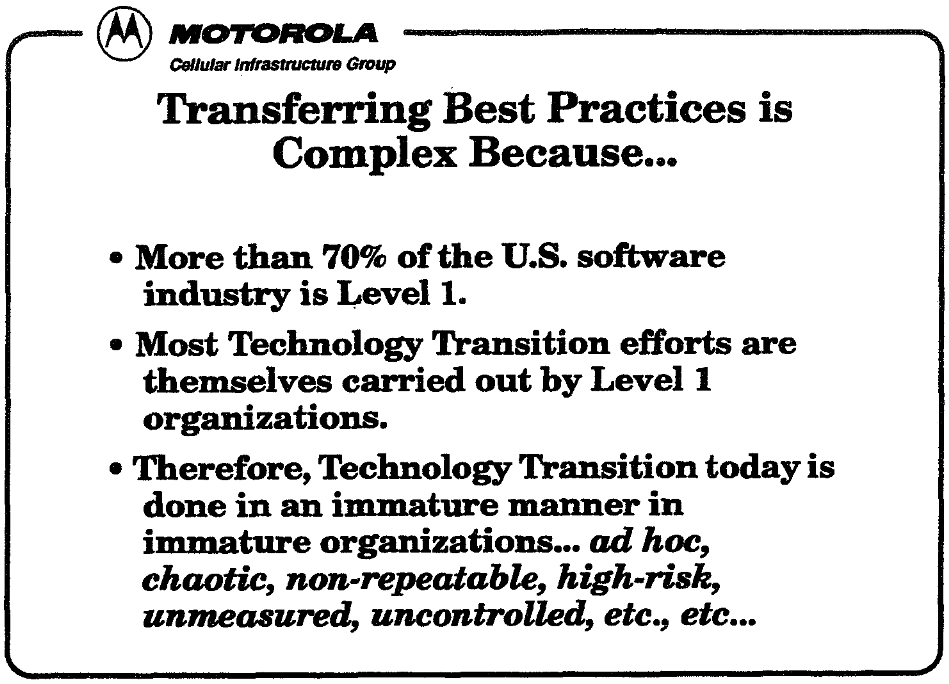


MOTOROLA
Cellular Infrastructure Group

Confirmations of Conventional Wisdom

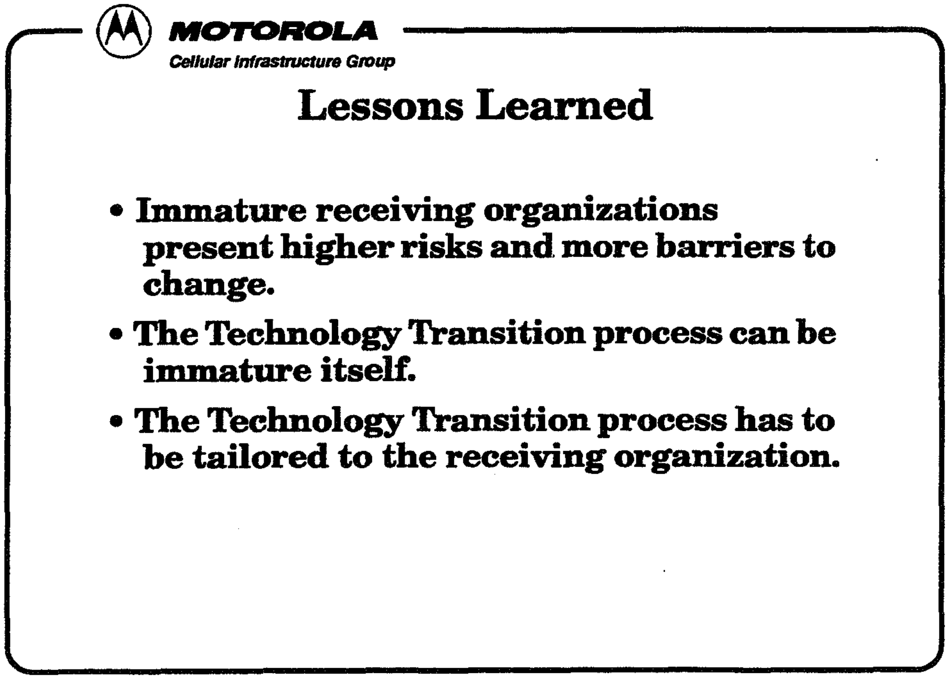
- Senior management sponsorship is needed, but that's not enough.
- New technologies diffuse best where there is a sense of urgency.
- Receiving organizations must provide resource support to assure success.

21st SEW Panel--12/3/96


MOTOROLA
Cellular Infrastructure Group

Transferring Best Practices is Complex Because...

- More than 70% of the U.S. software industry is Level 1.
- Most Technology Transition efforts are themselves carried out by Level 1 organizations.
- Therefore, Technology Transition today is done in an immature manner in immature organizations... *ad hoc, chaotic, non-repeatable, high-risk, unmeasured, uncontrolled, etc., etc...*


MOTOROLA
Cellular Infrastructure Group

Lessons Learned

- Immature receiving organizations present higher risks and more barriers to change.
- The Technology Transition process can be immature itself.
- The Technology Transition process has to be tailored to the receiving organization.

Appendix A: Workshop Attendees

Appendix A: Workshop Attendees

Adams, Carleen, TRW
Agresti, Bill W., MITRETEK Systems
Allen, Carmen L., Sandia National Laboratories
Anderson, Barbara, Jet Propulsion Lab
Anderson, Frances E., IIT
Angevine, Jim, ALTA Systems, Inc.
Ayers, Everett, Ayers Associates
Babchak, Joel, FAA Technical Center
Bailey, John, Software Metrics, Inc.
Barnette, James,DISA/JIEO/JEBE
Basili, Victor R., University of Maryland
Bassman, Mitchell J., Computer Sciences Corp.
Beall, Shelly, Social Security Administration
Becker, Shirley, Q-Labs, Inc.
Bellinger, Dwight Q., Consultant
Berry, Christine E., TRW
Bharadwaj, Ramesh, Kaman Sciences Corp.
Bhatia, Kiran, MITRETEK Systems
Binkley, Nita S., Treasury Financial Management Service
Bismut, Noemie A., Unisys/SATC
Blagmon, Lowell E., Naval Center For Cost Analysis
Blaney, Greg, NASA IV&V Facility
Blue, Velma D., DISA
Boland, Dillard, Computer Sciences Corp.
Bollman, John, Unisys Corp.
Bond, Walt, The Aerospace Corp.
Bozoki, George, Lockheed Martin Missiles & Space
Branigan, Christopher S., DISA/CFOS
Brugge, Lynda, Social Security Administration
Burke, Steve, Computer Sciences Corp.
Calavaro, Giuseppe F., Hughes Information Technology Corp.
Caldiera, Gianluigi, University of Maryland
Calhoun, Cynthia C., NASA IV&V Facility
Camaioni, Joseph, FAA
Cantone, Giovanni, University of Maryland
Carlson, Randall, NSWCDD
Celentano, Al, Social Security Administration
Centafont, Noreen, DoD
Cernell, Lori, NASA/KSC
Chiverella, Ron, Pennsylvania Blue Shield
Chu, Richard, Lockheed Martin (SMS)
Cockrell, Jake, University of Virginia
Coleman, Charles, Unisys Corp.
Condon, Steven E., Computer Sciences Corp.
Corbin, Genie, Social Security Administration
Corderman, Elizabeth, NASA/GSFC
Cuesta, Ernesto, Computer Sciences Corp.
Cusick, James, AT&T
Dandridge, Tondalya G., Treasury - Financial Management Service
Dane, Batia, GTE
Daniele, Carl J., NASA/LeRC
Darby, Clifton, Computer Sciences Corp.
Dawson, Jim, Bell Atlantic
DeMillo, Richard A., BELLCORE
Decker, William J., Computer Sciences Corp.
Derr, Patricia K., PRC, Inc.
Deutsch, Michael S., HITS
Dudash, Ed, Naval Surface Warfare Center
Duffy, Terry, McDonnell Douglas
Duvall, Lorraine, Kaman Sciences Corp.
Edelson, Robert, Jet Propulsion Lab
Elliott, Frank, Social Security Administration
Elliott, James, Lockheed Martin SMS
Ellis, Walter J., Software Process & Metrics
Evangelist, Michael, Florida International University
Evans, Roger M., Unisys Corp.
Ferguson, Frances, STeL
Fernandes, Vernon; Computer Sciences Corp.
Fike, Sherri, Ball Aerospace
Filmore, Tom, SETA Corp.
Fitz, Rhonda S., Azimuth, Inc.
Flynn, Margaret M., Treasury Financial Management Service
Forsythe, Ron, NASA/Wallops Flight Facility
Fountain, Elizabeth A., NASA/JSC
Franklin, Jude E., PRC, Inc.
Freeman, Peter A., Georgia Institute of Technology
Fuchs, Authur, NASA/GSFC
Futcher, Joseph M., Naval Surface Warfare Center
Gaddis, John B., VHTC Foundation
Gallman, Paula J., DISA
Galloway, Mary, Computer Sciences Corp.
Garris, Jason, Booz, Allen & Hamilton, Inc.
Gender, Paul W., Mantech International, Inc.
Getto, Gerhard, Daimler-Benz AG
Gibson, Jr., Richard G., American University

Gladden, Joycelyn M., Treasury - Financial Management Service
Glass, Robert L., The Software Practitioner
Glazer, Joel, Northrop Grumman Corp.
Godfrey, Sally, NASA/GSFC
Goel, Amrit L., Syracuse University
Goodenough, John, Software Engineering Institute
Gosnell, Barry, U.S. Army MICOM
Gotterbarn, Donald, George Washington University/Virginia
Grasso, Gayle A., SECON, Inc.
Gravitte, June A., Lockheed Martin SMS
Green, David S., Computer Sciences Corp.
Green, Scott, NASA/GSFC
Griffin, Robin, Lockheed Martin SMS
Gwynn, Thomas R., Computer Sciences Corp.
Halterman, Karen, NASA/GSFC
Hammer, Theodore F., NASA/GSFC
Hankinson, Al, Unisys/SATC
Hanna, Frank, Unisys Corp.
Hayes, Karen, Computer Sciences Corp.
Heasty, Richard, Computer Sciences Corp.
Heitmeyer, Connie, Naval Research Lab
Heller, Gerry H., Computer Sciences Corp.
Hermann, Brian, U.S. Air Force/KAFB
Hientz, Horst, Q-Labs GmbH
Hinchliffe, Nancy, Computer Sciences Corp.
Hirsch, Stephan J., DoD
Holmes, Glenda, [No Organization Registered]
Hopkins, Susan, DSCI
Howland, John C., EMC Corp.
Huffman, Lenore, Unisys/SATC
Hung, Chaw-Kwei, Jet Propulsion Lab
Isicoff, Richard, AlliedSignal Technical Services Corp.
Jay, Elizabeth M., NASA/GSFC
Jeletic, Jim, NASA/GSFC
Jeletic, Kellyann, NASA/GSFC
Jing, Yin, Computer Sciences Corp.
Jordano, Tony J., SAIC
Kacker, Raghu, NIST
Kassebaum, Kass, Consultant
Kay, Phyllis, Computer Sciences Corp.
Kelley, Ken, DISA
Kelly, John C., Jet Propulsion Lab
Kelly, Jr., Patrick, DISA
Kester, Rush W., Computer Sciences Corp.
Kim, Yong-Mi, Q-Labs, Inc.
Knight, John C.,University of Virginia
Kontio, Jyrki, University of Maryland
Kouchakdjian, Ara, Software Engineering Technology, Inc.
Kraft, Steve, NASA/GSFC
Kronisch, Mark I., U.S. Census Bureau
Kuhn, Rick, NIST
Landis, Linda C., Computer Sciences Corp.
Lane, Allan C., AlliedSignal Technical Services Corp.
Lane, Tricia P., U.S. Army DSMC
Lanubile, Filippo, University of Maryland
LeMire, Steven H., Northrop-Grumman Corp.
Letellier, Sandy, Treasury - Financial Management Service
Liebermann, Roxanne, U.S. Census Bureau
Lipsett, Bill, IRS
Liu, Jean C., Computer Sciences Corp.
Loesh, Bob E., Software Engineering Sciences, Inc.
Lott, Christopher M., BELLCORE
Lucas, Janice P., U.S. Treasury Department
Lydon, Tom, Raytheon
Lyle, William D.,The Analytic Sciences Corp.
Machak, Tom, BTG, Inc.
Maddox, Bill A., GDE Systems, Inc.
Mariano, Francisco C., IIT Research Institute
Martin, Tracey, Treasury - Financial Management Service
Matthews, Paul, BELLCORE
Maury, Jesse, Omitron, Inc.
McCanne, Randy, U.S. Air Force
McGarry, Frank E., Computer Sciences Corp.
McGuire, Eugene, American University
McIlwraith, Isabel, IRS
McSharry, Maureen, T.Rowe Price Associates, Inc.
McTavish, Charles J., NASA/GSFC
Melo, Walcelio L., CRIM
Mendonca, Manoel G., University of Maryland
Morgan, Elizabeth D., AlliedSignal Technical Services Corp.
Morusiewicz, Linda M., Computer Sciences Corp.
Murray, Albert, RAM Engineering Associates
Myers, Marguerite, Treasury Financial Management Service
Myers, Philip I., Computer Sciences Corp.
Neal, Ralph D., West Virginia University
Neil, Martin, City University of London
Nestlerode, Howard, Unisys Corp.

New, Ronald, Compu-Think
O'Donnell, Charlie, ECA, Inc.
O'Leary, Gerald, Treasury - Financial Management Service
O'Neill, Don, Consultant
Ohlsson, Niclas, Linkoping University, Sweden
Page, Gerald T., Computer Sciences Corp.
Pailen, William, Pailen-Johnson Associates, Inc.
Pajerski, Rose, NASA/GSFC
Panlilio-Yap, Nikki M., Lockheed Martin Corp.
Parra, Amy T., Computer Sciences Corp.
Pavnica, Paul, Treasury - Financial Management Service
Pendergrass, Vicki, NASA/GSFC
Perry, Howard, Computer Sciences Corp.
Pfister, Robin, NASA/GSFC
Pilch, Carol, GTE
Powers, Larry T., Unisys Corp.
President, John, DISA
Quann, Eileen S., Fastrak Training, Inc.
Ramsburg, M. C., DoD
Rauta, Constantin, Computer Sciences Corp.
Redding, John L., Defense Information Systems Agency
Reed, Natalie, FAA Technical Center
Regardie, Myrna L., Computer Sciences Corp.
Rico, David, SRA Corp.
Rinearson, Linda, GTE
Ritter, Sheila J., NASA/GSFC
Robbins, Jr., Henry H., Hughes Training, Inc.
Roberts, Sharon, ALTA Systems, Inc.
Robertson, Terry, CSC/TMG
Rodriguez, Mario, U.S. Air Force
Rodriguez, Roberto A., AlliedSignal Technical Services Corp.
Rombach, H.Dieter, University of Kaiserslauter
Rosenberg, Linda H., Unisys/SATC
Roy, Dan M., STP&P
Rudy, Jr., Paul L., Pennsylvania Blue Shield
Ruley, LaMont, NASA/GSFC
Russell, Christopher, Bureau of National Affairs, Inc.
Ryder, Regina, USDA
Sample, Keith, Lockheed Martin SMS
Samuels, George, Social Security Administration
Scheirer, Jeana M., Q-Labs, Inc.
Schneider, Laurie, General Sciences Corp./SAIC
Schneidewind, Norman, Naval Postgraduate School
Schuler, Mary P., NASA/LaRC
Schwartz, Benjamin L., Consultant
Schwarz, Henry, NASA/KSC
Schweiss, Robert, NASA/GSFC
Scott, Hester, ALTA Systems, Inc.
Seaman, Carolyn B., University of Maryland
Seamon, Dena C., [No Organization Registered]
Sharma, Jagdish, NOAA
Shin, Mi-Young, Syracuse University
Shull, Forrest, University of Maryland
Simpson, Brenda, SETA
Siwiec, Lenore S., ALTA Systems, Inc.
Slud, Eric V., University of Maryland
Smidts, Carol, University of Maryland
Smith, David, Computer Sciences Corp.
Smith, Donald, NASA/GSFC
Smith, George F., Space & Naval Warfare Systems Command
Smith, Len, Computer Sciences Corp.
Smith, Vivian A., FAA
Sohmer, Robert, RAM Engineering Associates
Soistman, Ed, Lockheed Martin
Sorumgard, Sivert, Norwegian Univeristy of Science & Tech.
Sparmo, Joe, NASA/GSFC
Squires, Burton E., Consultant
Stapko, Ruth, Unisys/SATC
Stark, Michael, NASA/GSFC
Sudermann, James E., NASA/KSC
Sullivan, Kevin J., University of Virginia
Swann, Mark, DoD
Sykes, Mari, Computer Sciences Corp.
Tepfenhart, William M., AT&T
Tesoriero, Roseanne, University of Maryland
Thomas, Bill, MITRE Corp.
Thompson, Sid, Unisys Corp.
Tittle, John G., Computer Sciences Corp.
Truong, Son H., NASA/GSFC
Ulery, Bradford T., MITRETEK Corp.
Valett, Jon, Q-Labs, Inc.
Valett, Susan, NASA/GSFC
VanMeter, Charlene L., DoD
Vargas, Sharon, Treasury - Financial Management Service
Waligora, Sharon R., Computer Sciences Corp.
Wallace, Dolores, NIST
Walsh, Chuck, RMS Associates
Warner, Kenneth N., TRW
Watson, Jim, NASA/LaRC
Wetherholt, Martha S., NASA/LeRC
Whisenand, Tom, Social Security Administration

Widmaier, James, DoD
Wiegand, Bob, NASA/GSFC
Willey, Allan L.,Motorola, Inc.
Wilson, Robert K., Jet Propulsion Lab
Wilson, William, Unisys/SATC
Wilson, William R., DoD
Wong, Raphael, U.S. Treasury
Woodyard, Charles E., NASA/GSFC
Wortman, Kristin, Computer Sciences Corp.
Wu, James, Treasury - Financial Management Service
Yassini, Siamak, NASA IV&V Facility
Zelkowitz, Marv, University of Maryland

Appendix B: Standard Bibliography of SEL Literature

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities. The *Annotated Bibliography of Software Engineering Laboratory Literature* contains an abstract for each document and is available via the SEL Products Page at http://fdd.gsfc.nasa.gov/selprods.html.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop,* August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop,* September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop,* September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study,* P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment,* T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3),* W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations,* K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment,* C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop,* November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation,* W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model,* A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop,* November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems,* J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November 1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature,* D. Kistler, J. Bristow, and D. Smith, November 1994

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-301, *Software Engineering Laborary (SEL) Database Organization and User's Guide (Revision 3)*, L. Morusiewicz, February 1995

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November 1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-94-003, *C Style Guide*, J. Doland and J. Valett, August 1994

SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994

SEL-94-005, *An Overview of the Software Engineering Laboratory*, F. McGarry, G. Page, V. R. Basili, et al., December 1994

SEL-94-006, *Proceedings of the Nineteenth Annual Software Engineering Workshop*, December 1994

SEL-94-102, *Software Measurement Guidebook (Revision 1)*, M. Bassman, F. McGarry, R. Pajerski, June 1995

SEL-95-001, *Impact of Ada in the Flight Dynamics Division at Goddard Space Flight Center*, S. Waligora, J. Bailey, M. Stark, March 1995

SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995

SEL-95-004, *Proceedings of the Twentieth Annual Software Engineering Workshop*, December 1995

SEL-95-102, *Software Process Improvement Guidebook (Revision 1)*, K. Jeletic, R. Pajerski, C. Brown, March 1996

SEL-96-001, *Collected Software Engineering Papers: Volume XIV*, October 1996

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., “Viewing Maintenance of Reuse-Oriented Software Development,” *IEEE Software*, January 1990

[13]Basili, V. R., "The Experience Factory and Its Relationship to Other Quality Approaches," *Advances in Computers*, vol. 41, Academic Press, Incorporated, 1995

[1]Basili, V. R., and J. Beane, “Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?,” *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[13]Basili, V. R., L. Briand, and W. L. Melo, *A Validation of Object-Oriented Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3443, UMIACS-TR-95-40, April 1995

[13]Basili, V. R., and G. Caldiera, *The Experience Factory Strategy and Practice*, University of Maryland, Computer Science Technical Report, CS-TR-3483, UMIACS-TR-95-67, May 1995

[9]Basili, V. R., G. Caldiera, and G. Cantone, “A Reference Architecture for the Component Factory,” *ACM Transactions on Software Engineering and Methodology,* January 1992

[10]Basili, V. R., G. Caldiera, F. McGarry, et al., “The Software Engineering Laboratory—An Operational Software Experience Factory,” *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, “Programming Measurement and Estimation in the Software Engineering Laboratory,” *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[12]Basili, V. R., and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994, pp. 58–66

[3]Basili, V. R., and N. M. Panlilio-Yap, “Finding Relationships Between Effort and Other Variables in the SEL,” *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, “Software Errors and Complexity: An Empirical Investigation,” *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, “Evaluating and Comparing Software Metrics in the Software Engineering Laboratory,” *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[13]Basili, V. R., M. Zelkowitz, F. McGarry, G. Page, S. Waligora, and R. Pajerski, "SEL's Software Process-Improvement Program," *IEEE Software*, vol. 12, no. 6, November 1995, pp. 83–87

[14]Basili, V. R., S. Green, O. Laitenberger, F. Shull, S. Sorumgard, and M. V. Zelkowitz, "*The Empirical Investigation of Perspective-Based Reading,*" University of Maryland, Computer Science Technical Report, CS-TR-3585, UMIACS-TR-95-127, December 1995

[14]Basili, V. R., "Evolving and Packaging Reading Technologies," P*roceedings of the Third International Conference on Achieving Quality in Software*, January 1996

[14]Basili, V. R., "The Role of Experimentation in Software Engineering: Past, Current, and Future," P*roceedings of the Eighteenth Annual Conference on Software Engineering (ICSE-18)*, March 1996

[14]Basili, V. R., G. F. Calavaro, G. Iazeolla, "Simulation Modeling of Software Development Processes," *7th European Simulation Symposium (ESS '95)*, October 1995

Bassman, M. J., F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA-GB-001-94, Software Engineering Program, July 1994

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, University of Maryland, Technical Report TR-3048, March 1993

[12]Briand, L. C., V. R. Basili, Y. Kim, and D. R. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 19–23, 1994, pp. 38–49

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[13]Briand, L., W. L. Melo, C. B. Seaman, and V. R. Basili, "Characterizing and Assessing a Large-Scale Software Maintenance Organization," *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, U.S.A., April 23–30, 1995

[11]Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

[12]Briand, L., S. Morasca, and V. R. Basili, *Defining and Validating High-Level Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3301, UMIACS-TR-94-75, June 1994

[13]Briand, L., S. Morasca, and V. R. Basili, *Property-based Software Engineering Measurement*, University of Maryland, Computer Science Technical Report, CS-TR-3368, UMIACS-TR-94-119, November 1994

[13]Briand, L., S. Morasca, and V. R. Basili, *Goal-Driven Definition of Product Metrics Based on Properties*, University of Maryland, Computer Science Technical Report, CS-TR-3346, UMIACS-TR-94-106, December 1994

[11]Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993

[14]Briand, L., Y. Kim, W. L. Melo, C. B. Seaman, V. R. Basili, "Qualitative Analysis for Maintenance Process Assessment," University of Maryland, Computer Science Technical Report, CS-TR-3592, UMIACS-TR-96-7, January 1996

[14]Briand, L., V. R. Basili, S. Condon, Y. Kim, W. L. Melo and J. D. Valett, "Understanding and Predicting the Process of Software Maintenance Releases," P*roceedings of the Eighteenth Annual Conference on Software Engineering (ICSE-18)*, March 1996

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering.* New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference.* New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. R. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[11]Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

McGarry, F., R. Pajerski, G. Page, et al., *Software Process Improvement in the NASA Software Engineering Laboratory,* Carnegie-Mellon University, Software Engineering Institute, Technical Report CMU/SEI-94-TR-22, ESC-TR-94-022, December 1994

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[12]Porter, A. A., L. G. Votta, Jr., and V. R. Basili, *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, University of Maryland, Technical Report TR-3327, July 1994

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[14]Seaman, C. B., V. R. Basili, "*Communication and Organization in Software Development: An Empirical Study,*" University of Maryland, Computer Science Technical Report, CS-TR-3619, UMIACS-TR-96-23, April 1996

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

[12]Seidewitz, E., "Genericity versus Inheritance Reconsidered: Self-Reference Using Generics," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark, M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications,* September 1993

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[13]Stark, M., and E. Seidewitz, "Generalized Support Software: Domain Analysis and Implementation," *Addendum to the Proceedings OOPSLA '94*, Ninth Annual Conference, Portland, Oregon, U.S.A., October 1994, pp. 8–13

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[13]Thomas, W. M., A. Delis, and V. R. Basili, *An Analysis of Errors in a Reuse-Oriented Development Environment*, University of Maryland, Computer Science Technical Report, CS-TR-3424, UMIACS-TR-95-24, February 1995

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS _92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[14]Waligora, S., J. Bailey, and M. Stark, "The Impact of Ada and Object-Oriented Design in NASA Goddard's Flight Dynamics Division," *Proceedings of the 13th Annual Washington Ada Symposium,* July 1996

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D.C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

[14]Zelkowitz, M. V., "Software Engineering Technology Infusion Within NASA," *IEEE Transactions On Engineering Management*, vol. 43, no. 3, August 1996

NOTES:

[1] This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2] This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3] This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4] This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5] This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6] This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7] This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8] This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9] This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10] This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11] This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

[12] This article also appears in SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994.

[13] This article also appears in SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995.

[14] This article also appears in SEL-96-001, *Collected Software Engineering Papers: Volume XIV*, October 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1996

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
Software Engineering Laboratory Series
Proceedings of the 21st Annual Software Engineering Workshop

5. FUNDING NUMBERS
Code 551

6. AUTHOR(S)
Flight Dynamics Systems Branch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS (ES)
Goddard Space Flight Center
Greenbelt, Maryland 20771

8. PEFORMING ORGANIZATION REPORT NUMBER
SEL-96-002

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS (ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
TM—1998–208617

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category: ~~82~~ 61
Report available from the NASA Center for AeroSpace Information,
7121 Standard Drive, Hanover, MD 21076-1320. (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Software Engineering Laboratory (SEL) is an organization sponsored by NASA/GSFC and created to investigate the effectiveness of software engineering technologies when applied to the development of application software.

The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

14. SUBJECT TERMS
Software Engineering Laboratory, Proceedings
Application software, Documentation

15. NUMBER OF PAGES
420

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL