NASA/TM-1998-208616

SOFTWARE ENGINEERING LABORATORY SERIES SEL-95-004

PROCEEDINGS OF THE TWENTIETH ANNUAL SOFTWARE ENGINEERING WORKSHOP

DECEMBER 1995

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Proceedings of the Twentieth Annual Software Engineering Workshop

November 29–30, 1995

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

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, Software Engineering Branch

The University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

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:

Software Engineering Branch
Code 552
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

What's Happening in the Software Engineering Laboratory?
Rose Pajerski, NASA/Goddard

The Empirical Investigation of Perspective-Based Reading
Vic Basili, University of Maryland

Porting Experience Factory Concepts to New Environments
Frank McGarry, Computer Sciences Corporation

S1-61

What's Happening in the Software Engineering Laboratory?

411597

18P.

360675

Rose Pajerski, Scott Green, and Donald Smith
Code 552, Software Engineering Branch
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771

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. The SEL consists of three member organizations: NASA/Goddard, the University of Maryland, and Computer Sciences Corporation. Throughout the SEL's history, its overall goal has remained the same: to improve the Division's software products and processes in a quantifiable manner.

Achieving this goal requires that each development and maintenance effort be viewed, in part, as a SEL experiment that examines a specific technology or builds a model of interest for use on subsequent efforts. The SEL has undertaken many technology studies while developing operational support systems for numerous NASA spacecraft missions. Data from over 120 software development projects in the organization have been collected and archived. From these data, the SEL has derived models of the development process and product and has conducted studies on the impact of new technologies.

This paper presents an overview of recent activities and studies in the SEL, using as a framework the SEL's organizational goals and experience-based software improvement approach. It focuses on two SEL experience areas: the evolution of the measurement program and an analysis of three generations of Cleanroom experiments.

Software Improvement Approach

The SEL's basic approach toward software process improvement is to first understand and characterize the process and product as they exist to establish a local baseline (Figure 1). Only then can new technologies be introduced and assessed (phase two) with regard to both process changes and product impacts. Typically, several studies and assessments are in progress at any one time, each with a duration of 1 to 3 years. The third phase of the SEL approach (packaging) synthesizes the results of the first two phases and feeds them back into the cycle for use by software engineers on subsequent projects. Experience packages include process-tailoring guidelines, training courses, tools, and guidebooks. The SEL's process improvement approach has proven very effective in the Flight Dynamics Division. The Division's software product has shown substantial improvements in error rates, cost, and development time. In 1994, the SEL received the IEEE Computer Society Award for Software Process Achievement and a Federal Technology Leadership Award for its application of these concepts in the FDD environment.

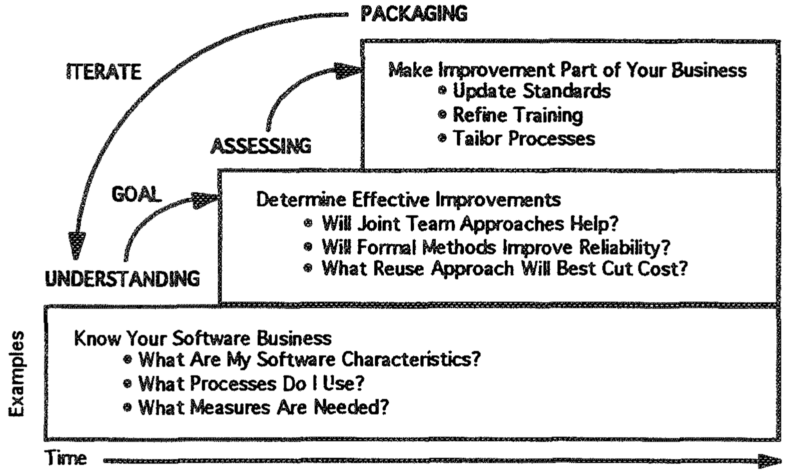

PACKAGING
ITERATE
ASSESSING
Make Improvement Part of Your Business
• Update Standards
• Refine Training
• Tailor Processes
GOAL
Determine Effective Improvements
• Will Joint Team Approaches Help?
• Will Formal Methods Improve Reliability?
• What Reuse Approach Will Best Cut Cost?
UNDERSTANDING
Know Your Software Business
• What Are My Software Characteristics?
• What Processes Do I Use?
• What Measures Are Needed?
Examples
Time

Figure 1. SEL Software Improvement Approach

SEL Goals

From its inception, the SEL has focused on both increasing reliability and reducing life cycle costs. Over the past 8 years, the SEL has achieved measured gains in both areas: reliability of delivered systems has increased threefold, and current mission support costs are half that of older systems. However, with increasing pressure to reduce "time to deploy," SEL goals now emphasize reducing development time as well as cost. Enabling process technologies have been selected, and analyses are underway to measure and assess their impact on both cost and schedule. In addition, a viable process improvement infrastructure must be maintained to realize these organizational goals. The four improvement goals (based on the 1994 SEL baseline) and the study areas/process technologies being investigated with each are as follows:

- By 1998, deliver systems 30% faster—Commercial-off-the-shelf (COTS) and reuse processes, testing approaches
- By 1998, reduce development cost per mission by 50%—COTS and reuse processes, testing approaches
- Maintain development error rate of 1 error per thousand lines of code—Cleanroom and reading techniques

- Maintain experience base—Testing characterization, maintenance baseline, measurement program

The remainder of this paper focuses on two of these activities: the evolution of the measurement program and the SEL's Cleanroom experiments.

Evolution of the Measurement Program

By collecting software metrics, the Software Engineering Laboratory has been able to provide a substantial level of support to projects within the Flight Dynamics Division. These services include building reliable software cost and schedule models, assisting in managing and predicting project performance, and understanding the impact that new technologies have on our process and products. Recently, however, changes in the FDD environment and the limitations of the existing data collection process and database structure have made it more difficult for the SEL to provide expected levels of support.

Recent changes to the development and operational environments include:

- Migration from mainframe to client-server architecture
- Compressed life cycles with overlapping phases and multiple builds
- Use of COTS and object-oriented technologies (OOT)

Another driver for change was the realization that the SEL's use of measurement had evolved and matured over the past 6 to 8 years. Different process measures were needed, especially in the areas of inspection and testing, to assess new approaches to these activities. Incremental development had become the norm, and managers wanted flexibility in grouping activity data into different phases rather than follow a strict waterfall life cycle. In addition to examining the measurement data, the SEL spent a substantial amount of effort examining the data collection process itself. Improvements were identified to address the following limitations:

- The data collection system did not collect effort by subsystem, thus creating problems in providing estimation planning support for single-subsystem development projects.
- Many incremental changes had been made to the data collection process over the years to support focused studies, and consequently the data collection system was becoming increasingly more difficult to maintain.
- The database structure required "tribal" knowledge to understand the relationships between projects, which created problems for researchers.
- The majority of entries were performed manually.
- The data collection system performed only a few ongoing checks, which resulted in large amounts of effort at project closeout.

In response, the SEL embarked upon an effort to reexamine the entire measurement program. The goals of this effort were to:

- Identify the measures that accurately reflect development, maintenance, and testing in the FDD.
- Develop a data collection system that captures those measures in the most efficient yet accurate way possible.
- Design a database structure that will house both new and old data in a format that is more intuitive for new users and is more flexible to accommodate future data collection needs.

The approach taken was to get input from all SEL member groups and then design, develop, and implement the new data collection system. Initially, a diverse group of FDD managers, University of Maryland researchers, and SEL staff gathered for a series of meetings to lay the groundwork for the effort. This group identified what was good and/or bad about the current data collection system and database; what changes they would like to see occur, and what additional data should be collected. The result of these meetings was an initial draft of the requirements for the new SEL data collection system.

Two groups were formed for the next phase. The first group was a smaller one that worked out the requirement details and finished the requirements document. The second group was the working group responsible for designing and implementing the data collection forms, data collection procedures, and database.

One of the more interesting items that was called out in the requirements and database design was a project-system-subsystem hierarchy, which provides the flexibility to view the data from many perspectives. This hierarchy solved several problems. First, it allows related projects to be grouped into a single system or project, making it easier for new database users who do not possess "tribal" knowledge of the data. Second, it stores data in the granularity needed for the creation of subsystem models and comparisons.

The collection and entry processes were examined in detail for potential improvements in processing time, work flow, and paper reduction. Some of the more interesting changes in these areas are to allow data form entry by project personnel using Word templates; to automate quality assurance (QA) checks with automatic data form entry into the database; and to set up repository tables for forms that fail QA checks until the discrepancies are resolved.

To summarize, this effort among all the SEL partners examined the entire measurement program and has led to:

- Defined measures that better represent the process and product—Deleted 50% of the original set of measures, added inspection data, and changed test measures for an overall 30% increase in measures collected.

- Improved the efficiency of collecting and quality assuring the data—Decreased the effort required to collect and enter data by 30%.
- Improved data retrieval—Used COTS tools extensively to diagram the database organization and improve access.

SEL Cleanroom Case Studies

Project Descriptions

Since the start of the SEL's investigation into the applicability of the methodology in 1988, four Cleanroom projects have been completed:

- Upper Atmosphere Research Satellite combined coarse/fine attitude determination and star identification systems, also known as the Attitude Cleanroom Methodology Experiment (ACME)
- Solar, Anomalous, and Magnetospheric Particle Explorer telemetry processor (SAMPEXTP)
- WIND/POLAR attitude ground support system (WINDPOLR)
- Solar and Heliospheric Observatory attitude ground support system (SOHOAGSS)

The SAMPEXTP and WINDPOLR efforts were conducted simultaneously and are considered as two data points under the SEL's second-generation Cleanroom process model. The SOHOAGSS, the latest project to be completed using the Cleanroom methodology, is the SEL's third-generation Cleanroom process model. A detailed analysis of the first- and second-generation systems was published in 1994 (Reference 1); this paper summarizes those results and provides a comparison with the most recent project (Reference 2).

During each project, an experiment team consisting of NASA/Goddard managers, SEL representatives, and a technology advocate was formed to monitor the overall process. Modifications were made to the process in real time as necessary. Also, specific data was collected at various points in the project life cycle for monitoring by the experimenter team, although this was done with as little impact as possible to the project team.

Experiment Goals

As its primary goal, the Cleanroom methodology emphasizes defect prevention rather than defect removal. It focuses on incrementally producing an error-free software product through processes that promote statistical quality control. The goal is to produce software with a high probability of zero defects and an operational measure of reliability.

The key elements of the methodology include an emphasis on human discipline in the development process, a mathematically based design approach, and a statistical testing approach based on anticipated operational usage. Development and testing teams are independent, and all development team activities are performed without on-line testing. Use of box structures, state machines, reading by stepwise abstraction, formal correctness demonstrations, and peer review are applied as necessary.

System development is performed through a pipeline of small increments to enhance concentration and permit parallel testing and development to occur. The mathematically based design approach and stepwise abstraction technique, in conjunction with emphasis on peer review, serve to ensure program correctness. The statistically oriented testing allows for reliability models to be used for quality assessment.

On the initial Cleanroom project (ACME), the SEL's primary goal was to attempt to increase software quality and reliability without incurring a negative cost impact. The SEL was also interested in contrasting characteristics of Cleanroom efforts with those of typical non-Cleanroom development efforts. A well-calibrated baseline for comparison existed that described a variety of process characteristics, including effort distribution, change rates, error rates, and productivity. This baseline represented a historical summary of a large number of previous SEL studies at the start of the SEL's examination of Cleanroom.

The goal of the second SEL Cleanroom case study was threefold. First, the SEL was interested in verifying the measures from the initial Cleanroom project by applying the methodology to another project of similar size and scope (SAMPEXTP). The initial effort indicated potential benefits for the Flight Dynamics Division, and additional supporting data would help pinpoint particular strengths and weaknesses. Second, the SEL wanted to verify the applicability of Cleanroom on a project substantially larger but more representative of the development environment (WINDPOLR). Third, the SEL was interested in impacts due to further process tailoring based on the initial study results and experiences.

The recently completed fourth project (SOHOAGSS) focused primarily on examining the scaling ability of the methodology. Analysis of the previous SEL Cleanroom studies had indicated greater success in applying the methodology to smaller (less than 50K developed lines of code (DLOC)) in-house development projects. However, typical ground system development efforts include the development of multiple utilities and subsystems and generally contain 100K to 200K DLOC. These projects are also usually staffed with contractor or joint Government/contractor teams, and earlier Cleanroom analysis of this type of project had yielded less promising results. As in the earlier studies, the SOHOAGSS project would be compared to previous SEL Cleanroom efforts and to the SEL baseline projects with respect to process, cost, and reliability.

Experiment Analysis

Product and process measures were continually examined to determine the impact of the tailored Cleanroom methodology. The SOHOAGSS project followed the Cleanroom approach that had evolved through the previous SEL Cleanroom projects, with two key changes: removal of the compilation restriction on developers, and scheduled meeting points between the Cleanroom experiment team and the project team.

In all previous Cleanroom projects, developers were responsible for generating and reviewing all code, but the compilation and configuration steps were conducted by the test teams. However, project teams had cited two significant issues relative to this process: First, much of the review and inspection time was focused on uncovering syntactical errors that would otherwise have been highlighted during component compilation. This reduced the amount of effort targeted toward uncovering logic and interface errors. Second, all compilation errors uncovered by the test team were reported back to the developers, who in turn would make corrections and redeliver. This consumed testing effort that otherwise could be spent executing and evaluating test cases.

The experiment team decided to modify the process on this project and allow developers to compile code before it was inspected and delivered to the testers. The earlier project teams had also stressed the need for more interaction with the experiment team to facilitate discussions involving unclear process steps and to obtain early feedback of experiment results. As a result, periodic meetings were scheduled with the SOHOAGSS project team during the design and early coding stages. The meetings resulted in valuable communications between the experiment team and project personnel; later in the project, however, meetings were held less frequently due to difficulties in scheduling and the lack of topics requiring discussion.

Figure 2 compares one key process element, activity effort distribution, across the three generations of Cleanroom projects. The figure indicates that the SOHOAGSS project continued the trend of increased design effort and reduced coding effort found in the earlier Cleanroom SEL studies. This highlights the project's reliance on well-understood component design and the criticality of successful design and code peer review. The comparison also verifies that, by adopting aspects of the Cleanroom methodology, developers were fundamentally changing their way of doing business.

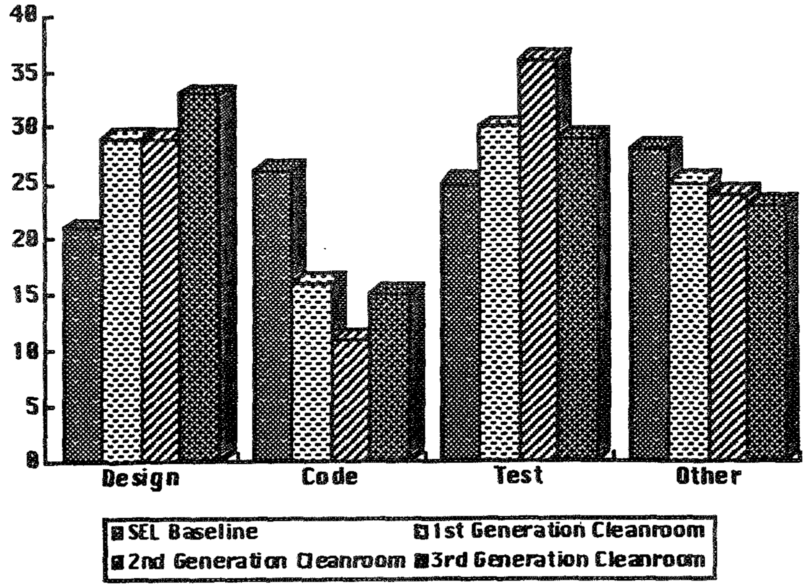


Figure 2. Effort Distribution by Activity (Percent of Total Effort)

Once a software component is placed under configuration control, SEL data is collected on every change and error correction made to the software. For non-Cleanroom projects, configuration control occurs following unit testing by the developer. On Cleanroom projects, software is controlled following the component peer inspections. Each error and change is classified in a variety of ways for analysis and comparison with other SEL projects. The errors recorded are primarily the result of independent system-level testing, although a small percentage of errors is uncovered by developers after the code has been controlled. Errors documented between the configuration control of the software and the operational release of the target system are classified as development errors; errors tracked after the operational release are classified as operational errors.

Figures 3a and 3b examine the error profiles exhibited by the SEL Cleanroom projects in development (errors per KDLOC) and during the first 2 years of mission operational support (errors per 100 KSLOC per year). The SEL baselines reflect different periods to provide a more representative comparison for each of the error types. The development rates on the Cleanroom projects are all below the 1989 baseline (when the SEL's Cleanroom research began), with the recent SOHOAGSS project measurably lower than all previous Cleanroom efforts. Since one chief goal of the project was to examine the scaling ability of the methodology in the environment, this result is of particular significance. Before the SOHOAGSS project, the benefits of the methodology for reducing development error rates had been seen only on the smaller SEL projects. Operationally, the successive generations of Cleanroom projects have resulted in a downward trend in measured error rates.

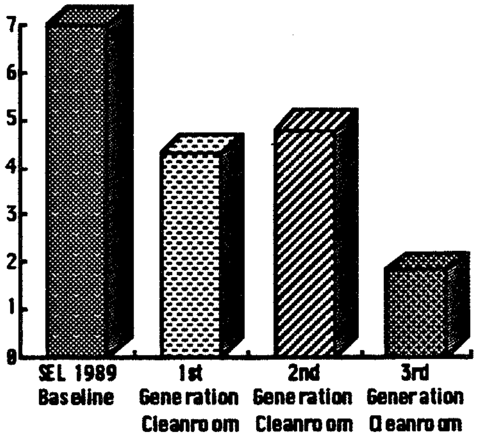


Figure 3a. Development Error Rates (Errors per KDLOC)

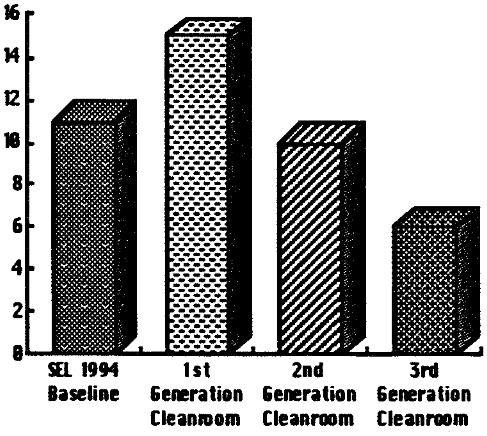


Figure 3b. Operational Error Rates (Errors per 100 KSLOC)

One of the original SEL Cleanroom goals was to improve the reliability of systems without negatively affecting cost. The cost of developing the initial SEL Cleanroom project, captured as team productivity in DLOC per day (Figure 4), actually showed measured improvement over the baseline. However, the second-generation projects dipped slightly below the baseline for non-Cleanroom projects. The SOHOAGSS project rebounded with a 33% increase over the baseline and actually showed a 60% increase over the similar second-generation WINDPOLR project.

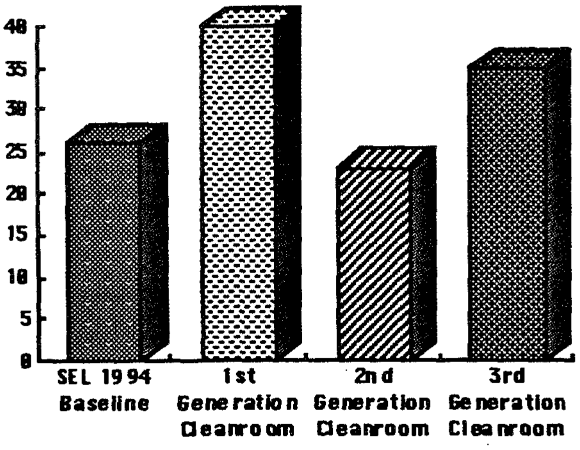


Figure 4. Productivity (SLOC per Day)

It is unclear to what extent various factors contributed to these favorable results. Candidate factors may include removal of the compilation restriction, previous Cleanroom experience by some development and test team members, high reuse percentages, and the application of lessons learned from previous SEL Cleanroom projects. Virtually all team members indicated a willingness to reapply the methodology on future SEL projects, as was true on all but one of the Cleanroom efforts. Project members also agreed that the reliance on a thorough and structured peer review process may ultimately make unit testing in the environment obsolete, even on non-Cleanroom projects.

Environment Impacts

The SEL's understanding of the impact of Cleanroom in the environment has matured such that significant elements of the tailored methodology are being incorporated as part of the general recommended approach to software development. These elements include:

- The formation of independent development and test teams
- An increased reliance on peer review and inspections as part of the verification process
- A decreased reliance on developer unit testing
- The inclusion of operational scenarios in acceptance testing

Report Summary

With these two efforts—the reengineering of the measurement program and the Cleanroom experiments—nearing completion, the SEL is well positioned to achieve its 1998 cost, schedule, and reliability goals. In the rapidly changing workstation environment, we are leveraging our experience in evolutionary process change, assessment, and improvement to achieve our goals. We have selected additional enabling process technologies in COTS usage, OOT, and testing approaches, and analyses are underway to measure and assess their impact on both cost and schedule. By following the SEL's experience-based improvement approach, other organizations can achieve similar results.

References

1. V. Basili and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994.

2. R. Coon, J. Golder, S. Green, and J. O'Neill, "*Solar and Heliospheric Observatory (SOHO) Mission Attitude Ground Support System (AGSS) Software Development History*," GSFC 552-FDD-95/026, November 1995.

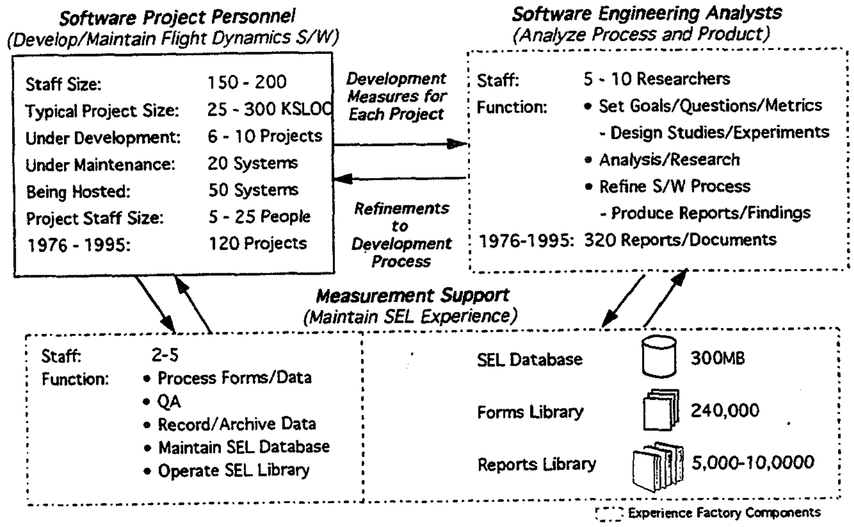
NASA/Goddard Space Flight Center

WHAT'S HAPPENING IN THE SOFTWARE ENGINEERING LABORATORY

Twentieth Annual
Software Engineering Workshop

Rose Pajerski
Scott Green
Don Smith

SEL ORGANIZATIONAL STRUCTURE



Software Project Personnel
(Develop/Maintain Flight Dynamics S/W)

Staff Size:	150 - 200
Typical Project Size:	25 - 300 KSLOC
Under Development:	6 - 10 Projects
Under Maintenance:	20 Systems
Being Hosted:	50 Systems
Project Staff Size:	5 - 25 People
1976 - 1995:	120 Projects

Development Measures for Each Project

Refinements to Development Process

Software Engineering Analysts
(Analyze Process and Product)

Staff: 5 - 10 Researchers

Function:
- Set Goals/Questions/Metrics
  - Design Studies/Experiments
- Analysis/Research
- Refine S/W Process
  - Produce Reports/Findings

1976-1995: 320 Reports/Documents

Measurement Support
(Maintain SEL Experience)

Staff: 2-5

Function:
- Process Forms/Data
- QA
- Record/Archive Data
- Maintain SEL Database
- Operate SEL Library

SEL Database	300MB
Forms Library	240,000
Reports Library	5,000-10,0000

Experience Factory Components

SEL GOALS

DELIVER SYSTEMS FASTER

1994 cycle time is 30% less than 1990 baseline - lower by 50% in 3 years

- COTS and reuse processes
- Testing approaches*

REDUCE COST

1994 cost (effort per mission) is 55% less than 1990 baseline - lower by 50% in 3 years

- COTS and reuse processes
- Testing approaches*

MAINTAIN (GOOD) RELIABILITY

1994 development error rate is 75% lower than 1985 - maintain/ lower slightly

- Cleanroom*
- Reading Techniques*

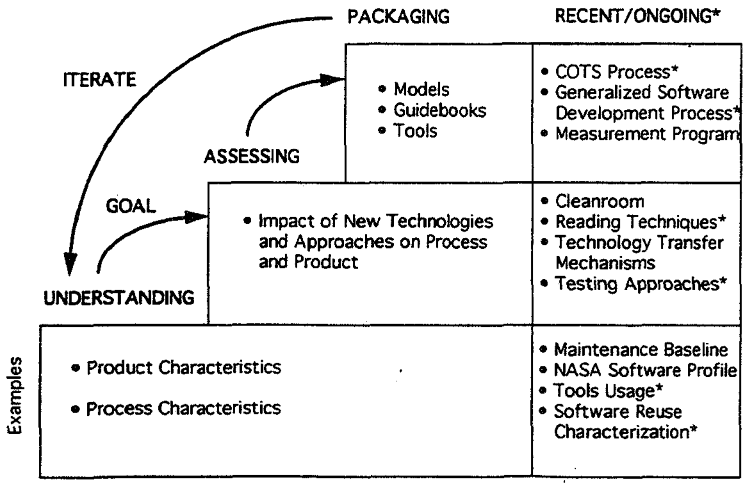
EVOLVE EXPERIENCE BASE

Product baseline computed in 1986, 1990, 1994 - continue to identify areas for improvement

- Testing Characterization*
- Maintenance Baseline
- Measurement Program*

*20th SEW Topics

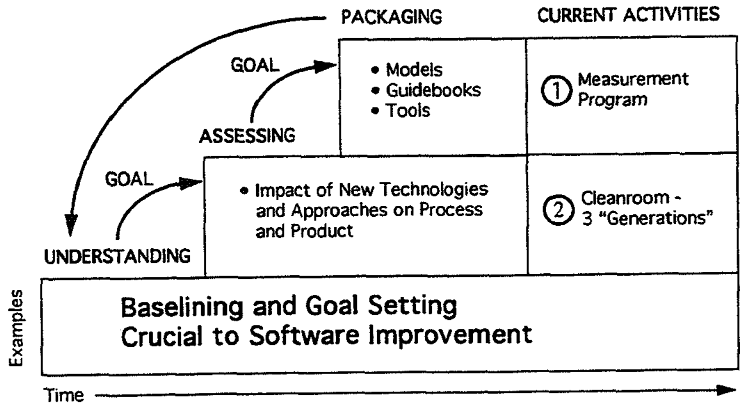
ACTIVITIES OVERVIEW



		RECENT/ONGOING*
PACKAGING	• Models • Guidebooks • Tools	• COTS Process* • Generalized Software Development Process* • Measurement Program
ASSESSING	• Impact of New Technologies and Approaches on Process and Product	• Cleanroom • Reading Techniques* • Technology Transfer Mechanisms • Testing Approaches*
UNDERSTANDING	• Product Characteristics • Process Characteristics	• Maintenance Baseline • NASA Software Profile • Tools Usage* • Software Reuse Characterization*

Experience-based Concept Accommodates Change

ADAPTING TO CHANGE



PACKAGING

CURRENT ACTIVITIES

GOAL

ASSESSING

- Models
- Guidebooks
- Tools

① Measurement Program

GOAL

UNDERSTANDING

- Impact of New Technologies and Approaches on Process and Product

② Cleanroom - 3 "Generations"

Baselining and Goal Setting Crucial to Software Improvement

Examples

Time

ADAPTING TO CHANGE - EXAMPLES IN THE SEL

- Infrastructure: Evolution of Measurement Program
  - Drivers for change
  - Approach
  - Observations
  -
- Multiple Case Study: Cleanroom Process Evolution
  - Study goals
  - Results on latest (3rd). generation project

① MEASUREMENT PROGRAM EVOLUTION

- Operational systems are changing
  - Client-server architecture/multiple platforms
  - Generalized software/COTS usage
  - Data collected no longer matched process
    - ➤ Overlapping phases/many small builds
    - ➤ Independent testing
- Use of measurement maturing
  - Re-evaluate measures
  - Flexibility to link data in many ways

Environmental Changes and Need for Flexibility Were Key Drivers

APPROACH

- Identify measurement data requirements
  - Start from scratch
  - Base on user (Managers and Researchers) needs
- Identify data collection process improvements
  - Transition from paper to on-line
  - Maintain QA steps
  - Base on user (Developers and Data Support Staff) needs
- Design/revise forms, database, process, and reports
  - New forms in use
  - New database created - being populated with old data
  - New processes being prototyped

MEASUREMENT PROGRAM CHANGES

To Data Content

- Project hierarchies
- Product/release information (Partial context data)
- Estimates/actuals by subsystem/application
- Schedule information (By phase and/or build)
- Includes inspection and testing data
- Integrates development and maintenance data

To Data Processing

- On-line forms templates
- Automated tools tied to CM process
- Streamlined forms tracking and discrepancy notification
- Plan to automate data transfer to database

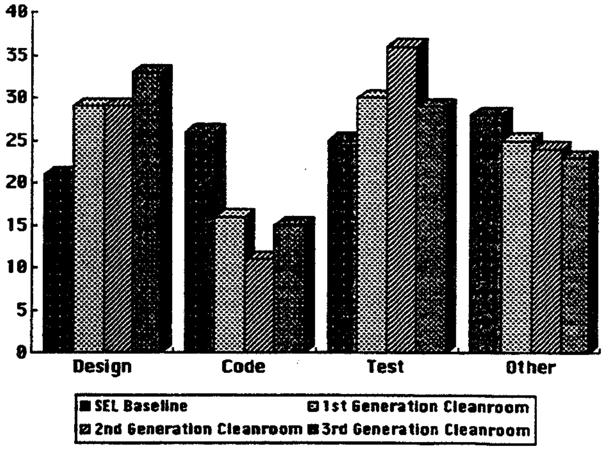
OBSERVATIONS

- Overall 30% increase in measures collected
  - Deleted 50% of original set, added inspection/test measures
  - Collection and entry process support reduced by 30%
  -
- Issue of "Context" data still open
  - Product-related elements now in database
  - Meaningful process analysis still requires subjective data/ personal interview
  -
- Database and related COTS tools are very powerful
  - No need for custom-built systems
  - Flexibility now linked with tools
  -

② SEL CLEANROOM CASE STUDIES

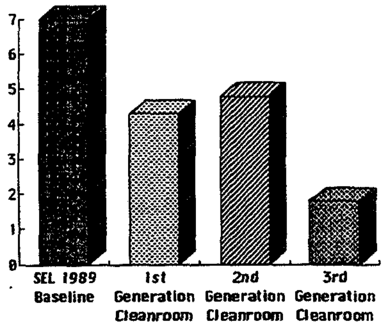
	1st Generation (~1990) (1 Project)	2nd Generation (~1992) (2 Projects)		3rd Generation (~1995) (1 Project)
Software Size	40K DLOC	21K DLOC	150K DLOC	140K DLOC
Team Size (Developers/Testers)	3/2	4/2	14/4	10/3
Process Changes	• Variation of Mills' Methodology • Higher Design Effort • Combined State Machines with Structured Design • Separate Development and Test Teams • Developers Did Not Compile	• Similar to 1st • More Training • Used Box Structure Design Technique		• Developers Compiled Code Before Transfer to Testers • Understanding of Cleanroom Process Greater (Culture)
Results	• Better Reliability • Better Productivity	• Mixed - Smaller Project Showed Similar Improvement to 1st; Larger Project Did Not		→

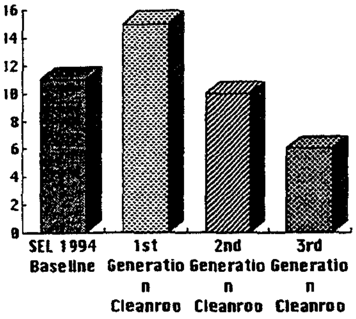
PROCESS IMPACT - EFFORT BY ACTIVITY



Relative Effort Tradeoff Between Design and Code for Cleanroom Projects

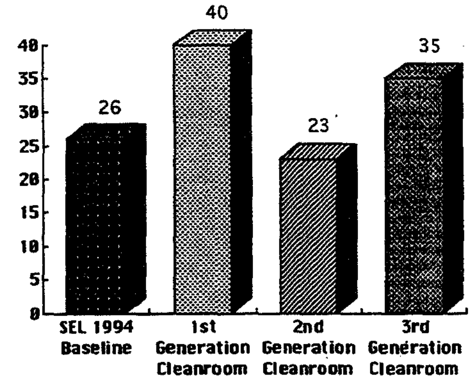
PRODUCT IMPACT - ERROR PROFILES


Development Errors (per KSLOC)


Operational Errors (per 100 KSLOC per Year)

PRODUCTIVITY

(Source Lines of Code per Day)



Cleanroom Process Elements:

- Compiler Restriction Removed
- Inspection Replaces Unit Test
- More Interaction With Process Analysts

Cleanroom Process Elements Are Now Mature and Being Packaged as SEL Standard

The Empirical Investigation of Perspective-Based Reading

Victor R. Basili[1], Scott Green[2], Oliver Laitenberger[3],
Forrest Shull[1], Sivert Sørumgård[4], Marvin V. Zelkowitz[1]

[1] Computer Science Department/
Institute for Advanced Computer Studies
University of Maryland, College Park, MD, 20742
{basili, fshull, mvz}@cs.umd.edu

[2] NASA Goddard Space Flight Center
Code 552.1
Greenbelt, MD, 20771
scott.green@gsfc.nasa.gov

[3] AG Software Engineering
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany
laitenbe@informatik.uni-kl.de

[4] The Norwegian Institute of Technology
The University of Trondheim
UNIT/NTH-IDT
O.S. Bragstads plass 2E
Trondheim, N-7034
Norway
sivert@idt.unit.no

This work was supported in part by NASA grant NSG-5123 and UMIACS.

Abstract

We consider reading techniques a fundamental means of achieving high quality software. Due to the lack of research in this area, we are experimenting with the application and comparison of various reading techniques. This paper deals with our experiences with Perspective-Based Reading (PBR), a particular reading technique for requirements documents. The goal of PBR is to provide operational scenarios where members of a review team read a document from a particular perspective (e.g., tester, developer, user). Our assumption is that the combination of different perspectives provides better coverage of the document than the same number of readers using their usual technique.

To test the efficacy of PBR, we conducted two runs of a controlled experiment in the environment of the National Aeronautics and Space Administration / Goddard Space Flight Center (NASA/GSFC) Software Engineering Laboratory (SEL), using developers from the environment. The subjects read two types of documents, one generic in nature and the other from the NASA domain, using two reading techniques, PBR and their usual technique. The results from these experiments, as well as the experimental design, are presented and analyzed. When there is a statistically significant distinction, PBR performs better than the subjects' usual technique. However, PBR appears to be more effective on the generic documents than on the NASA documents.

1. Introduction

The primary goal of software development is to generate systems that satisfy the user's needs. However, the various documents associated with software development (e.g., requirements documents, code and test plans) often require continual review and modification throughout the development lifecycle. In order to analyze these documents, reading is a key, if not *the* key technical activity for verifying and validating software work products. Methods such as inspections (Fagan, 1976) are considered most effective in removing defects during development. Inspections rely on effective reading techniques for success.

Reading can be performed on all documents associated with the software process, and can be applied as soon as the documents are written. However, except for reading by step-wise abstraction (Linger, 1979) as developed by Harlan Mills, there has been very little research focused on the development of reading techniques. Most efforts have been associated with the methods (e.g., inspections, walk-throughs, reviews) surrounding the reading technique. In general, techniques for reading particular documents, such as requirements documents or test plans, do not exist. In cases where techniques do exist, the required skills are neither taught nor practiced. In the area of programming languages, for example, almost all effort is spent learning how to *write* code rather than how to *read* code. Thus, when it comes to reading, little exists in the way of research or practice.

In the Software Engineering Laboratory (SEL) environment, we have learned much about the efficacy of reading and reading-based approaches through the application and evaluation of methodologies such as Cleanroom. We are now part of a group (ISERN[1]) that has undertaken a research program to define and evaluate software reading techniques to support the various review methods for software development.

In this paper, we use the following convention to differentiate a "technique" from a "method": A technique is a series of steps, producing some desired effect, and requiring skilled application. We define a method as a management procedure for applying techniques.

1.1 Experimental Context: Scenario-Based Reading

In our attempt to define reading techniques, we established several goals:

- The technique should be associated with the particular document (e.g., requirements) and the notation in which the document is written (e.g., English text). That is, it should fit the appropriate development phase and notation.
- The technique should be tailorable, based upon the project and environment characteristics. If the problem domain changes, so should the reading technique.
- The technique should be detailed, in that it provides the reader with a well-defined process. We are interested in usable techniques that can be repeated by others.
- The technique should be specific in that each reader has a particular purpose or goal for reading the document and the procedures support that goal. This can vary from project to project.
- The technique should be focused in that a particular technique provides a particular coverage of the document, and a combination of techniques provides coverage of the entire document.
- The technique should be studied empirically to determine if and when it is most effective.

To this end, we have defined a set of techniques, which we call proactive process-driven scenarios, in the form of algorithms that readers can apply to traverse the document with a particular emphasis. Because the scenarios are focused, detailed, and specific to a particular emphasis or viewpoint, several scenarios must be combined to provide coverage of the document.

We have defined an approach to generating a family of reading techniques based upon operational scenarios, illustrated in Figure 1. An operational scenario requires the reader to first create an abstraction of the product, and then answer questions based on the abstraction. The choice of abstraction and the types of questions asked may depend on the document being read, the problem history of the organization or the goals of the organization.

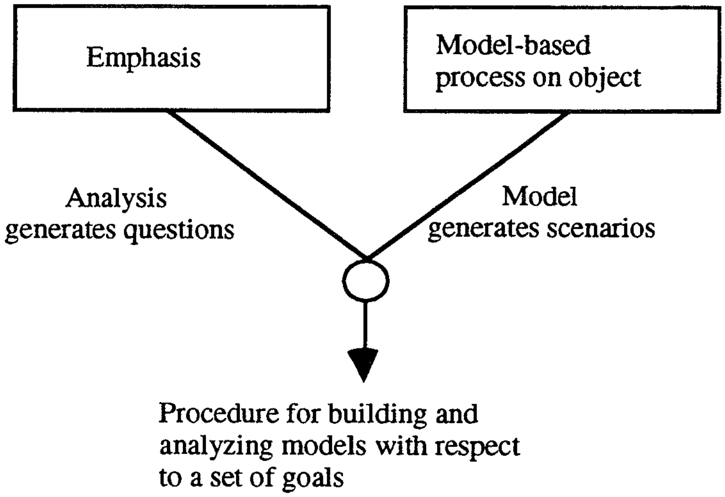

Emphasis
Model-based process on object
Analysis generates questions
Model generates scenarios
Procedure for building and analyzing models with respect to a set of goals

Figure 1. Building focused, tailored reading techniques.

So far, two different scenario-based reading techniques have been defined for requirements documents: perspective-based reading and defect-based reading.

Defect-based reading was the subject of an earlier set of experiments in this series. Defect-based reading was defined for reading SCR (Software Cost Reduction) style documents (Heninger, 1980), and focuses on different defect classes, e.g., missing functionality and data type inconsistencies. These create three different scenarios: data type consistency, safety properties, and ambiguity/missing information. An experimental study (Porter, 1995) was undertaken to analyze defect-based reading, ad hoc reading and checklist-based reading to evaluate and compare them with respect to their effect on defect detection rates. Major results were that (1) scenario readers performed better than ad hoc and checklist readers with an improvement of about 35%, (2) scenarios helped reviewers focus on specific defect classes but were no less effective at detecting other defects, and that (3) checklist reading was no more effective than ad hoc reading.

However, the experiment discussed in this paper is concerned with an experimental validation of perspective-based reading, and so we treat it in more detail in the next section.

1.2 Perspective-Based Reading

Perspective-based reading (PBR) focuses on the point of view or needs of the customers or consumers of a document. In this type of scenario-based reading, one reader may read from the point of view of the tester, another from the point of view of the developer, and yet another from the point of view of the user of the system. To provide a proactive scenario, each of these readers produces some physical model which can be analyzed to answer questions based upon the perspective. The team member reading from the perspective of the tester would design a set of tests for a potential test plan and answer questions arising from the activities being performed. Similarly, the team member reading from the perspective of the developer would generate a high level design, and the team member representing the user would create a user's manual. Each scenario is focused on one perspective. The assumption is that the union of the perspectives provides sufficient coverage of the document but does not cause any particular reader to be responsible for everything.

This work on PBR was conducted within the confines of the NASA/GSFC Software Engineering Laboratory. The SEL, started in 1976, has been developing technology aimed at improving the process of developing flight dynamics software within NASA/GSFC. This class of software is typically written in any of several programming languages, including FORTRAN, C, C++, and Ada. Systems can range from 20K to 1M lines of source code, with development teams of up to 15 persons working over a one to two year period.

Assume we embed these requirements reading scenarios in a particular method. It then becomes the role of the method to determine which scenarios to apply to the document, how many readers will play each role, etc. This could be done by assuming, as entry criteria, that the method has available to it the anticipated defect class distribution, together with knowledge of the organization's ability to apply certain techniques effectively. Note that embedding focused reading techniques in a method such as inspections provides more meaning to the "team" concept. That is, it gives the readers different views of the document, allowing each of the readers to be responsible for their own view, with the union of the readers providing greater coverage than any of the individual readers.

Consider, as an example, the procedure for a reader applying the test-based perspective:

> Reading Procedure: For each requirement, make up a test or set of tests that will allow you to ensure that the implementation satisfies the requirement. Use your standard test approach and test criteria to make up the test suite. While making up your test suite for each requirement, ask yourself the following questions:
>
> 1. Do you have all the information necessary to identify the item being tested and to identify your test criteria? Can you make up reasonable test cases for each item based upon the criteria?
>
> 2. Is there another requirement for which you would generate a similar test case but would get a contradictory result?
>
> 3. Can you be sure the test you generated will yield the correct value in the correct units?
>
> 4. Are there other interpretations of this requirement that the implementor might make based upon the way the requirement is defined? Will this effect the test you made up?
>
> 5. Does the requirement make sense from what you know about the application and from what is specified in the general description?

These five questions form the basis for the approach the test-based reader will use to review the document.

We developed two different series of experiments for evaluating scenario-based techniques. The first series of experiments are aimed at discovering if scenario-based reading is more effective than current practices. This paper's goal is to analyze perspective-based reading and the current NASA SEL reading technique to evaluate and compare them with respect to their effect on fault detection effectiveness. It is expected that other studies will be run in different environments using the same artifacts where appropriate. A second series, to be undertaken later, will be used to discover under which circumstances each of the various scenario-based reading techniques is most effective.

1.3 Experimental Plan

Our method for evaluating PBR was to see if the approach was more effective than the approach people were already using for reading and reviewing requirements specifications. Thus, it assumes some experience in reading requirements documents on the part of the subjects. More specifically, the current NASA SEL reading technique (SEL, 1992) had evolved over time and was based upon recognizing certain types of concerns which were identified and accumulated as a set of issues requiring clarification by the document authors, typically the analysts and users of the system.

To test our hypotheses concerning PBR, a series of partial factorial experiments were designed, where subjects would be given one document and told to discover defects using their current method. They would then be trained in PBR and given another document in order to see if their performance improved. We were initially interested in several outcomes:

1. Would individual performances improve if each individual used one of the PBR (designer, tester, user) scenarios in order to find defects?

2. If groups of individuals (such as during an inspection meeting) were given unique PBR roles, would the collection of defects be different than if each read the document in a similar way?

3. Are there characteristic differences in the class of defects each scenario uncovered?

While we were interested in the effectiveness of PBR within our SEL environment, we were also interested in the general applicability of the technique in environments different from the flight dynamics software that the SEL generally builds. Thus two classes of documents were developed: a domain-specific set that would have limited usefulness outside of NASA, and a generic set that could be reused in other domains.

For the NASA flight dynamics application domain, two small specifications derived from an existing set of requirements documentation were used. These specification documents, seeded with classes of errors common to the environment, were labeled NASA_A and NASA_B. For the generic application domain, two requirements documents were developed and seeded with known classes of errors. These applications included an automated parking garage control system, labeled PG, and an automated bank teller machine, labeled ATM.

1.4. Structure of this Paper

In section 2, we discuss how we developed a design for the experiment outlined above. Major issues concerning constraints and threats to validity are described in order to highlight some of the tradeoffs made. We also include a short overview of how the experiment was actually carried out.

Section 3 presents the statistical analysis of the data we obtained in the experiment. The section examines individual results and team results. In each of these parts, we look at the results from both experiment runs, within documents and within domains.

Section 4 is an interpretation of the results of the experiment, but without the rigor of a formal statistical approach. The presentation is again divided into individual results and team results, with concentration on what effect the differences between the two runs of the experiment had in terms of results.

Section 5 summarizes our experiences regarding designing and carrying out the experiment.

2. Design of the Experiment

In this section, we discuss various ways of organizing the individual subjects and the instrumentation of the experiment to test various hypotheses. Two runs of the experiment were conducted. Due to the experiences gained in the initial run, some modifications were introduced in its replication. Differences between the two runs of the experiment will be pointed out where appropriate.

For both experiments, the population was software developers from the NASA SEL environment. The selection of subjects from this sample was not random, since everyone in the population could not be expected to be willing or have opportunity to participate. Thus, all subjects were volunteers, and we accepted everyone who volunteered. Nobody participated in both runs of the experiment.

2.1 Hypotheses

We formulated our main question in the form of the following two hypotheses, where H_0 is the null-hypothesis and H_a is the alternative hypothesis:

H_0 *There is no significant difference in the defect detection rates of teams applying PBR as compared to teams using the usual NASA technique.*

H_a *The defect detection rates of teams applying PBR are significantly higher as compared to teams using the usual NASA technique.*

Our hypotheses are focused on the performance of teams, but we will also analyze the results for the individual performance of the subjects. We make no assumptions at this level regarding the validity of the hypotheses when changing important factors such as subjects, and documents. The constraints relevant for this particular experiment will be explicitly discussed throughout this section, as will the generalizability of the results of the experiment.

2.2 Factors in the Design

In designing the experiment, we had to consider what factors were likely to have an impact on the results. Each of these factors will cause a rival hypothesis to exist in addition to the hypotheses we mentioned previously. The design of the experiment has to take these factors, called *independent variables*, into account and allow each of them to be separable from the others in order to allow for testing a causal relationship to the defect detection rate, the *dependent variable* under study.

Below we list the independent variables, which we identify according to how they can be manipulated. Some of them can be controlled during the course of the experiment, while some are strictly functions of time, and still others are not even measurable.

- **Controllable variables:**

  - **Reading technique:** We have two alternatives: One is the technique we have developed, PBR, and the other is the technique currently used for requirements document review in the NASA SEL environment, which we refer to as the "usual" technique.
  - **Requirements documents:** For each task to be carried out by the subjects, a requirements specification is handed out to be read and reviewed. The document will presumably have an impact on the results due to differences in size, domain and complexity.
  - **Perspective:** For PBR, a subject can take one of three perspectives as previously described: Designer, Tester or User.

- **Measurable variables:**

  - **Replication:** This nominal variable is not one we can manipulate, but we need to be aware of its presence because there may be differences in the data from the two experiment runs that may be the result of changes to documents, training sessions or experimental conditions.
  - **Round within the replication:** For each experiment, every subject is involved in a series of treatments and tasks or observations. The results from similar tasks may differ depending on when they take place.

- **Other factors identified:**

  - **Experience:** The experience of each subject is likely to have an impact on the defect detection rate.
  - **Task sequence:** Reading the documents in a sequence may have an influence on the results. This may be a learning effect due to the repetitive reading of multiple documents.
  - **Environment:** The particular environment in which the experiment takes place may have an impact on how well the subjects perform. In this experiment, this effect cannot be separable from effects due to replication.

There will also be other factors present that may have an impact on the outcome of the experiment, but that are hard to measure and control. These will be discussed in Section 2.5. This section will also cover the last two factors mentioned above: Task Sequence (in the literature referred to as "effects due to testing") and Environment.

2.3 Constraints and Limitations.

In designing the experiment we took into account various constraints that restrict the way we could manipulate the independent variables. There are basically two factors that constrain the design of this experiment:

- **Time:** Since the subjects in this experiment are borrowed from a development organization, we could not expect to have them available for an indefinite amount of time. This required us to make the experiment as time-efficient as possible without compromising the integrity of the design.
- **Subjects:** For the same reasons as stated above, we could not expect to get as many subjects as we would have liked. This required us to be cautious in the design and instrumentation in order to generate as many useful data points as possible.

Specifically, we knew that we could expect to get between 12 and 18 subjects for two days on any run of the experiment.

Another factor that we had to deal with is that we had to provide some potential benefit to the subjects since their organization was supporting their participation. Training in a new approach provided some benefit for their time. This had an impact on our experimental design because we had to treat people equally as far as the training they received.

2.4 Choosing a Design

Due to the constraints, we found that constructing real teams of (three) reviewers to work together in the experiment would take too much time for the resulting amount of data points. This decision was supported by similar experiments (Parnas, 1985) (Porter, 1995) (Votta, 1993), where the team meetings were reported to have little effect; the meeting gain was outweighed by the meeting loss. However, the team is an important unit in the review process, and PBR is team-oriented in that each reviewer has a responsibility that is not shared by other reviewers on the team. Thus our reviewers did not work together in teams during the course of the experiment. Instead we conducted the experiment based on individual tests, and then used these individual results to construct hypothetical teams after the experiment was completed.

The tasks performed by the subjects consisted of reading and reviewing a requirements specification document, and recording the identified defects on a form. The treatments, which had the purpose of manipulating one or more of the independent variables, were aimed at teaching the subjects how to use PBR. There were four possible ways of arranging the order of tasks and treatments for a group of subjects:

1. Do all tasks using the usual technique.
2. Do pre-task(s) with the usual technique, then teach PBR, followed by post-task(s) using PBR.
3. Start by teaching PBR, then do some tasks with the PBR technique, followed by tasks using the usual technique.
4. Start by teaching PBR, then do all tasks using PBR.

Option 3, where the subjects first use PBR and then switch to their usual technique, was not considered an alternative because their recent knowledge in PBR may have undesirable influences on the way they apply their usual technique. The opposite may also be true, that their usual technique has an influence on the way they apply PBR, but that is a situation we cannot control because the subjects already know their usual technique. Thus, this becomes more a problem in terms of external validity.

All documents reviewed by a subject must be different. If a document was reviewed more than once by the same subject, the results would be disturbed by the subject's non-erasable knowledge about defects found in previous readings. This meant that we had to separate the subjects into two groups - one reading the first document and one reading the second in order to be able to compare a PBR and a usual reading of a document.

Based on the constraints of the experiment, each subject would have time to read and review no more than four documents: two from the generic domain, and two from the NASA domain. In addition, we needed one sample document from each domain for training purposes. We ended up providing the following documents:

- **Generic:**
  - Automatic teller machine (ATM) - 17 pages, 29 seeded defects.
  - Parking garage control system (PG) - 16 pages, 27 seeded defects.

- **NASA:**
  - Flight dynamics support (A) - 27 pages, 15 seeded defects
  - Flight dynamics support (B) - 27 pages, 15 seeded defects

- **Training:**
  - Video rental system - 14 pages, 16 seeded defects
  - NASA sample - 9 pages, 6 seeded defects

Since we have sets of different documents and techniques to compare, it became clear that a variant of factorial design would be convenient for this experiment. Such a design would allow us to test the effects of applying both of the techniques on both of the relevant documents. We found that a full factorial design would be inappropriate for two reasons. First, a full factorial design would require some subjects to apply the ordering of techniques that we previously argued against. Secondly, such a design seemed hard to conduct because it would require each subject to use all three perspectives at some point. This would require an excessive amount of training, and perhaps even more important, the perspectives would likely interfere with each other, causing an undesirable learning effect.

The use of control groups to assess differences in documents and learning effect appeared to bear an unreasonable cost, since the use of such groups would decrease the remaining number of data points available for analyzing the difference between the techniques. The low number of data points might result in data that would be heavily biased due to individual differences in performance. Based on the cost and the fact that previous related experiments (Porter, 1995) showed that effects of learning were not significant, we chose not to use control groups. This decision also made the experiment more attractive in terms of getting subjects, since they would all receive the same amount and kind of training.

	Group 1	Group 2	
	D T U	D T U	
NASA technique	Training	Training	First day
	NASA A	NASA B	
	Training	Training	
	ATM	PG	
PBR technique	Teaching of PBR		Second day
	Training	Training	
	PG	ATM	
	Training	Training	
	NASA B	NASA A	

Figure 2. Design of the experiment.

We blocked the design on technique, perspective, document and reading sequence in order to get an equal distribution of the values of the different independent variables. Thus we ended up with two groups of subjects, where each group contains three subgroups, one for each perspective (see Figure 2). The number of subjects was about the same for the two experiments (12-14 subjects).

2.5 Threats to Validity

Threats to validity are factors beyond our control that can affect the dependent variables. Such threats can be considered unknown independent variables causing uncontrolled rival hypotheses to exist in addition to our research hypotheses. One crucial step in the experimental design is to minimize the impact of these threats.

We have two different classes of threats to validity: threats to *internal* validity and threats to *external* validity. Threats to internal validity constitute potential problems in the interpretation of the data from the experiment. If the experiment does not have a minimum internal validity, we can make no valid inference regarding the correlation between variables. On the other hand, the level of external validity tells us nothing about whether the data is interpretable, but is an indicator of the generalizability of the results. Depending on the external validity of the experiment, the data can be assumed to be valid in other populations and settings.

The following five threats to internal validity (Campbell, 1963) are discussed in order to reveal their potential interference with our experimental design:

- **History:** We need to consider what the subjects did between the pretests and posttests. In addition to receiving a treatment where they were taught a new reading technique, there may have been other events outside of our control that had an impact on the results. The subjects were instructed not to discuss the experiment or otherwise do anything between the tests that could cause an unwanted effect on the results.
- **Maturation:** This is the effect of processes taking place within the subjects as a function of time, such as becoming tired or bored. But it may also be intellectual maturation, regardless of the experimental events. For our experiment, the likely effect would be that tests towards the end of the day tend to get worse results than they would normally. We provided generous breaks between sessions to suppress this effect.
- **Testing:** Getting familiar with the tests may have effects on subsequent results. This threat has several components, including becoming familiar with the specifications, the technique, or the testing procedures. We tried to overcome unwanted effects by providing training sessions before each test where the subjects could familiarize themselves with the particular kind of document and technique. Also, the subjects received no feedback regarding their actual defect detection success during the experiment, as this would presumably increase the learning effect. Related experiments have reported that effects due to repeated testing are not significant (Porter, 1995).
- **Instrumentation:** These effects are basically due to differences in the way of measuring scores. Our scores were measured by two people independently, and then discussed in order to resolve any disagreement consistently. Thus this effect is not relevant to us.
- **Selection:** Subjects may be assigned to their treatment groups in various ways. In our case there was a difference between the two experiment runs. In the first one, the subjects were assigned roles for PBR based on their normal work in the NASA environment in order to match roles as closely as possible. This was only minimally successful since the sample was not an even mix of people representing the various roles. However, for the replication, the subjects were randomized. Thus effects due to selection may be somewhat relevant for the first experiment, but not for the replication. Since PBR assumes the reviewers in a team use their usual perspectives, the random assignment used in the experiment would presumably lead to an underestimation of the improvement caused by PBR.

Another threat to validity is the possibility that the subjects ignore PBR when they are supposed to use it. In particular, there is a danger that the subjects continue to use their usual technique. This need not be the result of a deliberate choice from the subject, but may simply reflect the fact that people unconsciously prefer to apply existing skills with which they are familiar. The only way of coping with this threat is to provide enhanced training sessions and some sort of control or measure of conformance to the assigned technique.

Threats to external validity imply limitations to generalizing the results. The experiment was conducted with professional developers and with documents from an industrial context, so these factors should pose little threat to external validity. However, the limited number of data points is a potential problem which may only be overcome by further replications of the experiment. Other threats to external validity pertinent to the experimental design include (Campbell, 1963):

- **Interaction of testing and treatment:** A pretest may affect the subject's sensitivity of the experimental variable. Both of our groups receive similar pretests and treatments, so this effect may be of concern to us.
- **Interaction of selection and treatment:** Selection biases may have different effects due to interaction with the treatment. One factor we need to be aware of is that all our subjects were volunteers. This may imply that they are more prone to improvement-oriented efforts than the average developer - or it may indicate that they consider the experiment an opportunity to get away from normal work activities for a couple of days. Thus, the effects can strike in either direction. Also, all subjects had received training in their usual technique, a property that developers from other organizations may not possess.
- **Reactive effects:** These effects are due to the experimental environment. Here we have a difference between the two runs of the experiment. In the initial run, the testing was done in the subjects' usual work environment. The subjects received their training in groups, and then returned to their own workspace for the test. For the replication, the experiment was conducted in an artificial setting away from the work environment, similar to a classroom exercise. This may influence the external validity of the experiment, since a non-experimental environment may cause different results.

There are also a number of other possible but minor threats. One of these is the fact that the subjects knew they were part of an experiment. They knew that the purpose of the experiment was to compare reading techniques, and they probably were able to surmise our expectations with respect to the results even if not stated explicitly. However, these aspects are difficult to eliminate in experiments where subjects are trained in one technique while the comparison technique is assumed to be known in advance. A design where they receive equal training in two techniques would be more likely to hide these effects.

2.6 Preparation and Conduction

We wanted the two experiment runs to be as similar as possible in order to avoid difficulties in combining the resulting data, but some changes between the runs were still necessary. We began preparing for the second run by reviewing all documents and forms in order to improve them from an experimental viewpoint. We had some comments from the first experiment run that were helpful in this process. The changes were minor, and most were directed towards language improvement. We changed the seeded defects in three places in one of the generic documents due to a refined and deeper insight into what we would consider a defect. There were some changes to the forms, scenarios and defect classification as well, but again the changes were made to make the documents easier to use and understand.

For the NASA documents, the changes were more fundamental. For the first experiment run, comments from the participants indicated that the documents were too large and complex. We decided to make them shorter and simpler for the second experiment run. As a side effect of this change, the total number of defects in the NASA documents was reduced. However, the types and distribution of seeded defects remained similar.

The basic schedule for conducting the experiment remained unchanged. Each experiment run lasted for two whole work days, with one day off in-between. The number and order of document reviews were also the same for both experiments, but the time allowed for each review was modified. For the first experiment run, the maximum time for one document was three hours. However, for the generic documents, only one person used more than two hours (140 minutes), so under the more controlled environment of the second experiment run, we felt safe lowering the maximum time to two hours.

Another important change resulted from the comments we received from the first experiment run, regarding the training sessions. The initial run included training sessions only for the generic documents, but the subjects felt training for the NASA documents was warranted as well. Therefore in the second experiment run, we had training sessions before each document review. For this purpose we generated two sample documents that were representative of the NASA and generic domains.

After the second run of the experiment, we marked all reviews with respect to their defect detection rate. This was measured as the percentage of the seeded defects that was found by each reviewer. We did not consider any other measures such as false positives. Based on the defects found by the reviewers, we also refined our understanding of the defects present in the set of documents. After several iterations of discussion and re-marking, we arrived at a set of defect lists that were considered representative of the documents. Since these lists were slightly different from the lists that were used in the first experiment, we re-marked all the reviews from the first experiment in order to make all results consistent.

3. Statistical Analysis

We ran the experiment twice, in November 1994 (hereafter referred to as the "1994 experiment") and in June 1995 (hereafter referred to as the "1995 experiment"). In the 1994 experiment, we had twelve subjects read each document, six using the usual technique and six using PBR. The six using PBR were distributed equally among the three perspectives. In the 1995 experiment, we had thirteen subjects who read each document, although a fourteenth volunteer unfamiliar with the NASA domain also read the generic documents only.

After the two experiment runs, we have a substantial base of observations from which to draw conclusions about PBR. This task is complicated, however, by the various sources of extraneous variability in the data. Specifically, we identify four other variables (besides the reading technique) which may have an impact on the detection rate of a reviewer: the experiment run within which the reviewer participated, the problem domain, the document itself, and the reviewer's experience.

We attempted to measure reviewer experience via questionnaires used during the course of the experiment: a subjective question asked each reviewer to rate on an ordinal scale his or her level of comfort using such documents, and objective questions asked how many years the reviewer had spent in each of several roles (analyst, tester, designer, manager). However, for any realistic measurement scale, most reviewers tended to clump together toward the middle of the range, with relatively few outliers in either direction. Thus we seem to have a relatively homogeneous sample with respect to this variable. While good from an experimental viewpoint, this unfortunately means that our data set does not allow for a meaningful test of the effect of reviewer experience, and we are forced to defer an investigation of the interaction between reading technique and experience until such time as we can get more data points. For this reason, reviewer experience will not appear as a potential effect in any of our analysis models.

Technique, experiment run, and document are represented by nominal-scale variables used in our models, where appropriate. The domain is taken into account by performing a separate analysis for each of the generic and NASA problem domains. However, we are also careful to note that there are variables that our statistical analysis cannot measure. Perhaps most importantly, an influence due to a learning effect would be hidden within the effect of the reading technique. The full list of these threats to validity is found in Section 2.5, and any interpretation of results must take them into account.

Section 3.1 presents the details of the effect on individual scores. Section 3.2 presents the analysis strategy for team data. Finally, Section 3.3 takes an initial look at the analysis with respect to the reviewer perspectives. In each section, we present the general analysis strategy and some details on the statistical tests, followed by the statistical results and some interpretation of their meaning. We address the significance of our results taken as a whole in Section 4.

3.1 Analysis for Individuals

Although it was not part of our main hypothesis, which focuses on team coverage, we wanted to see if the difference in focus between the usual technique and PBR would have some effect on individual detection rates. We therefore went through an analysis of individual scores.

We were also careful, however, to test for effects from sources of variation other than the reading technique. For this reason, our analysis proceeds in a "bottom-up" manner. That is, we begin with several small data sets that we know to be homogeneous. Each session of the experiment was run under controlled conditions to eliminate differences within the sessions that might have an effect on reviewers' detection rates; the scores of reviewers reading the same document within the same replication are therefore comparable. Thus we begin our analysis with homogeneous data sets (4 documents - 2 NASA and 2 generic - over 2 runs, so 8 in total) which we will use as the primary building blocks of our analysis.

Starting from these data sets, we looked for features in common between the data sets. We identified subsets of the data which were expected to be more homogeneous than the data as a whole; the aim was to exploit this homogeneity to achieve stronger statistical results. For example, we took into account the fact that all of the detection rates for each reviewer are highly correlated, but we also identified other such blocks (e.g., the data for each problem domain within the experiment). As we looked at larger data sets in order to draw more general conclusions, we also took pains to make sure that the data within each set were still comparable. Figure 3 illustrates the direction of our analysis, and includes the sizes of the data sets.

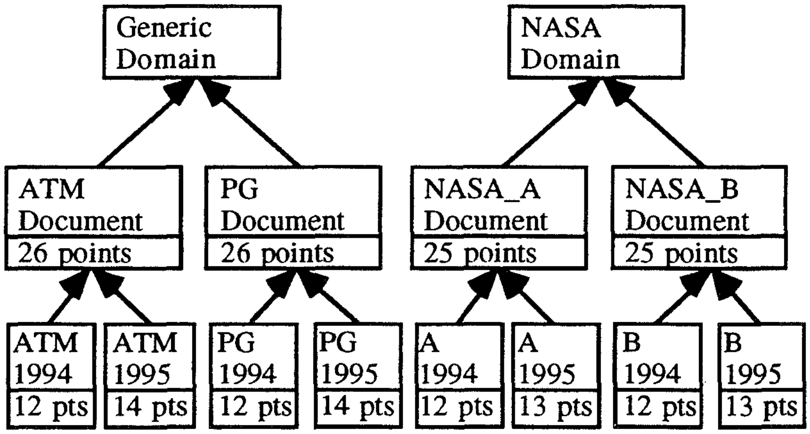


Figure 3. Breakdown of the statistical analysis, with number of data points.

3.1.1 Analysis Strategy Within Documents

Our initial analysis examined each document used in the experiment for significant differences in performance based on the use of reading technique. We used the ANOVA test since we were testing a model of the effects containing multiple potential sources of variation. To begin with, our model of the effects contained a nominal variable to signify the reading technique used (usual or PBR).

The data for each document is composed of the independent data sets from the two experiment runs, and so it was necessary to be alert to the possibility that changes from one run to the next could have an impact on the reviewers' detection rates. For both of the pairs of documents, we combined the data for the document and introduced a nominal variable (with two levels: 1994 and 1995) into our model to describe the experiment run in which the reviewer read the document.

We measured the lack of fit error (an estimate of the error variance) for the model on each document. In no case was there a significant lack of fit error, so it did not seem likely that we could gain any better fit to the data by introducing variations on the variables, such as testing for interaction effects (SAS, 1989).

We also tested whether each of the variables independently was significant (i.e., whether the effect of each variable, apart from the other variables in the model, had a significant effect on reviewer detection rate).

The ANOVA test makes a number of assumptions, which we were careful to fulfill: The dependent variable is measured on a ratio scale, and the independent variables are nominal. Observations are independent. The values tested for each level of the independent variables are normally distributed (we confirmed this with the Shapiro-Wilk W Test). Also, the test assumes that variance between samples for each level of the independent variables is homogeneous. However, we note that the test is robust against violations of this last assumption for data sets such as ours in which the number of subjects in the largest treatment group is no more than 1.5 times greater than the number of subjects in the smallest (Hatcher, 1994). The test also assumes that the sample must be obtained through random sampling; this is a threat to the validity of our experiment, as we must rely on volunteers for our subjects (see Section 2.5, "Selection" and "Interaction of selection and treatment").

3.1.2 Results Within Documents

In our case the hypotheses of the ANOVA test take the following form:

> **H_0:** The specified model (which contains variables to signify the experiment run and reading technique) has no significant power in predicting the value of the dependent variable (detection rate).
>
> **H_a:** The model as a whole is a significant predictor of detection rate.
>
> **Level of significance:** $\alpha = 0.05$

The ANOVA test also allows testing the effect of each individual variable.

The Least Squares Means (LSM) of the detection rates for reviewers using each of the techniques are given in Table 1, followed by the results of the tests for significance. The LSM values in effect allow an examination of the means for the groups using each of the reading techniques while holding the difference due to experiment run constant. This is followed by the p-values resulting from the statistical tests for significance; a p-value of less than 0.05 provides evidence that either the whole model or the individual variable is a significant predictor of detection rate and are indicated in boldface. The R^2 value for the model is also included as a measure of the amount of variation in the data that is accounted for by the model.

For all documents except NASA_B, the LSM detection rate for PBR reviewers is slightly higher than for reviewers using their usual technique. However, only for the ATM document was the difference statistically significant. For all other documents, reviewers using the two techniques did roughly the same, and any differences between their average scores can be attributed to random effects alone. Both NASA documents had a very significant effect due to experiment run, which was not surprising, given the large changes made to improve the documents between runs; however, there was also a significant and unexpected effect due to experiment run for the PG document as well. The significance of such differences due to experiment run is addressed in Section 4.

Document	PBR LSM	USUAL LSM	Whole Model p-value	Technique p-value	Replicat-ion p-value	R^2
ATM	30.8	21.4	0.0904	**0.0316**	0.6299	0.19
PG	26.8	24.5	**0.0457**	0.5977	**0.0174**	0.24
NASA_A	36.8	26.6	**0.0001**	0.1516	**0.0001**	0.56
NASA_B	28.3	34.5	**0.0021**	0.5044	**0.0005**	0.43

Table 1. Effects on individual scores for each document.

3.1.3 Analysis Strategy Within Domains

The second level of detail which we analyzed was the level of problem domains. That is, we examined what trends could be observed within the generic documents or within the NASA documents, while realizing that such trends may not necessarily apply across such different domains. For each domain, we tested whether each reviewer scored about the same when reviewing documents with PBR as when using the usual technique, or if there was in fact a significant effect due to reading technique.

To accomplish this, we made use of the MANOVA (Multivariate ANOVA) test with repeated measures, an extension of the ANOVA which measures effects across multiple dependent variables (here, the scores on each of the two documents) with longitudinal data sets (i.e. data sets in which each subject is represented by multiple data points).

The domain data sets contain two scores for each subject, one for each document within the domain. Although repeated measures tests usually refer to multiple treatments over time, here we treat the scores on each document as the scores from repeated treatments, which we distinguish with the nominal variable "Document". We divide the reviewers into two groups, and use another nominal variable in order to distinguish to which group each reviewer belonged: Group I applied PBR to Document A and the usual technique to Document B, and Group II read the documents in the opposite fashion. If the interaction between these two variables is significant, we can conclude that the reading technique a reviewer applied to each document had a significant effect on the reviewer's detection rate. If the interaction is not significant, then reviewers tended to perform about the same on the two documents, regardless of the technique applied to each. Aside from reading technique and document, we again want to account for any significant effects due to the experiment run, and also test for interaction effects between this variable and the others.

The MANOVA test with Repeated Measures makes certain assumptions about the data set. As with the ANOVA test, we again fulfill requirements about the measurement scales of the dependent and independent variables, the independence of observations, and the underlying distribution of the sample. We have the same threat to validity resulting from the assumption of random samples as was discussed for the ANOVA test. However, it is also assumed that the dependent-variable covariance matrix for a given treatment group should be equal to the covariance matrix for each of the remaining groups. Fortunately, the type I error rate is relatively robust against typical violations of this assumption; however, the power of the test is somewhat attenuated (Hatcher, 1994).

3.1.4 Results Within Domains

Using the data from each of the documents within a domain, we use the MANOVA test to detect how reviewer rates change from one document to the next, and attribute these changes to factors in our model. As we did with the ANOVA test, we test whether each of the variables in our model (the documents themselves, the reading technique used on each document, the experiment run, and all appropriate interactions) are significant predictors of the change in detection rates.

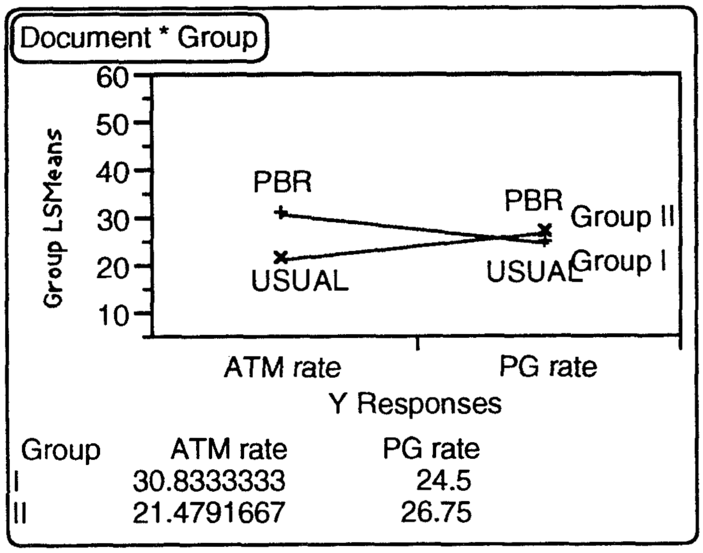
H_0: The specified variable has no significant effect in predicting scores across the two documents.
H_a: The variable is a significant predictor of scores across the documents.
Level of significance: $\alpha = 0.05$

The results are summarized in Table 2, where each column gives the p-value for each of the effects. A p-value of less than 0.05 provides an indication that the variable is a significant predictor of the change in reviewer detection rates across documents, and appears in bold. The effect due to the reading technique is measured indirectly by the "Group" variable: Group I read Document A with the PBR technique and Document B with the usual technique; Group II read the documents in the reverse fashion. As can be seen from the "Document" column, there was no significant difference between the mean detection rates for the two documents within a domain. Crossed terms represent tests for interaction effects; for example, the column labeled "Document * Replication" tests if the mean difference in reviewers' scores on each of the documents was significantly effected by the experiment run in which they took part. Thus, even though the NASA documents were changed drastically between runs, because the two documents were roughly comparable in difficulty within both experiment runs, there is no significant effect here for the NASA domain. Within the generic domain, reviewers in the 1994 experiment did slightly better on the PG document than the ATM, while reviewers in the 1995 experiment did slightly worse on the PG document relative to the ATM; while the differences average out when the two runs are combined, the effect still shows up as a significant interaction in the MANOVA test.

Domain	**Document**	**Document * Replication**	**Document * Group**	**Document * Replication * Group**
Generic	0.7810	**0.0298**	**0.0056**	0.5252
NASA	0.9137	0.7672	0.5670	0.5337

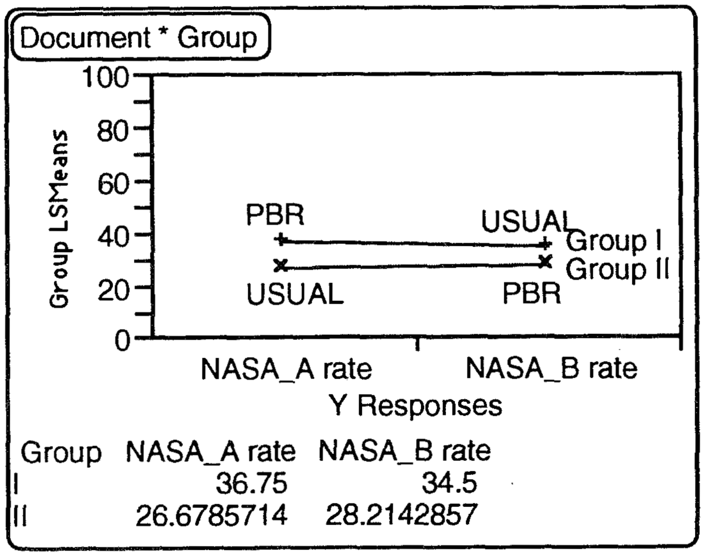
Table 2. Effects on individual scores within domains.

Graphs of Least Squares Means are presented in figures 4a and 4b as a convenient way of visualizing the effects of the interaction between document and reading technique. For the generic domain, it can be seen that reviewers in each group on average scored higher with PBR than with the usual technique, taking into account the other effects in the model. In the NASA domain, reviewers in each group scored about the same on both documents, regardless of the technique used. Note that the interaction for the generic domain is significant, providing evidence that reading technique does in fact have an impact on detection rates.



Group	ATM rate	PG rate
I	30.8333333	24.5
II	21.4791667	26.75

Figure 4a. Interaction between group and technique for the generic domain.



Group	NASA_A rate	NASA_B rate
I	36.75	34.5
II	26.6785714	28.2142857

Figure 4b. Interaction between group and technique for the NASA domain.

3.2 Analysis for Teams

3.2.1 Analysis Strategy for Teams

In this section, we return to investigating our primary hypothesis concerning the effect of PBR on inspection teams. The analysis was complicated by the fact that the teams were composed after the experiment's conclusion, and so any grouping of individual reviewers into a team is somewhat arbitrary, and does not signify that the team members actually worked together in any way. The only real constraint on the makeup of a team which applied PBR is that it contain one reviewer using each of the three perspectives; the non-PBR teams can have any three reviewers who applied their usual technique. At the same time, the way in which the teams are composed has a very strong effect on the team scores, so an arbitrary choice can have a significant effect on the test results.

For these reasons, we used a permutation test to test for differences in team scores between the techniques. An informal description of the test follows.

First, since there are differences between the experiment runs, we will compose teams only with reviewers from within the same run; we therefore treat the two experiment runs separately. Results from the individual scores showed that the domains are very different, but the documents within a domain are of comparable difficulty; thus, we compare reviewer scores on documents within the same domain only. We again categorize reviewers into one of two groups, as we did for the analysis within domains for individual scores, depending on which technique they applied to which document. Let us say the reviewers in Group I applied PBR to Document A and their usual technique to Document B, where Document A and Document B represent the two documents within either of the domains. We can then generate all possible PBR teams for Document A and all possible non-PBR teams for Document B, and take the average detection rate of each set. This ensures that our results are independent of any arbitrary choice of team members, but because the data points for all possible teams are not independent (i.e., each reviewer appears multiple times in this list of all possible teams), we cannot run simple statistical tests on these average values. For now, let us call these averages A_I and B_I. We can then perform the same calculations for Group II, in which reviewers applied their usual technique to Document A and PBR to Document B, in order to obtain averages A_{II} and B_{II}. The test statistic

$$(A_I - B_I) - (A_{II} - B_{II})$$

then gives us some measure of how all possible PBR teams would have performed relative to all possible usual technique teams.

Now suppose we switch a reviewer in Group I with someone from Group II. The new reviewer in Group I will be part of a PBR team for document A even though he used the usual technique on this document, and will be part of a usual technique team for Document B even though he applied PBR. A similar but reversed situation awaits the reviewer who suddenly finds himself in Group II. If the use of PBR does in fact improve team detection scores, one would intuitively expect that as the PBR teams are diluted with usual technique reviewers, the average score will decrease, even as the average score of usual technique teams with more and more PBR members is being raised. Thus, the test statistic computed above will decrease. On the other hand, if PBR does in fact have no effect, then as reviewers are switched between groups the only effect will be due to random effects, and team scores may improve or decrease with no correlation with the reading technique of the reviewers from which they are formed. So, let us now compute the test statistic for all possible permutations of reviewers between Group I and Group II, and rank each of these scenarios in decreasing order by the statistic. If the scenario in which no dilution has occurred appears toward the top of the list (in the top 5%) we will conclude PBR does have a beneficial effect on team scores, since every time the PBR teams were diluted with non-PBR reviewers they tended to perform somewhat worse relative to the usual technique teams. However, should the non-diluted scenario appear toward the middle of the list, then this is clear evidence that every successive dilution had only random effects on team scores, and thus that reading technique is not correlated with team performance.

Note that this is meant to be only a very rough and informal description of the intuition behind the test; the interested reader is referred to Edington's *Randomization Tests* (Edington, 1987).

3.2.2 Results for Teams

The use of the permutation test allows us to formulate and test the following hypotheses:

H_0: The difference between average scores for PBR and usual technique teams is the same for any random assignment of reviewers to groups.

H_a: The difference between average scores for PBR and usual technique teams is significantly higher when the PBR teams are composed of only PBR reviewers and the usual technique teams are composed of only usual technique reviewers.

Level of significance: $\alpha = 0.05$ (that is, we reject H_0 if the undiluted teams appear in the top 5% of all possible permutations between groups)

The results are summarized in Table 3. P-values which are significant at the 0.05-level appear in boldface. For example, twelve reviewers read the generic documents in the 1994 experiment; there are 924 distinct ways they can be assigned into groups of 6. The group in which there was no dilution had the 61st highest test statistic, corresponding to a p-value of 0.0660.

Domain/ Replication	Number of Group Permutations Generated	Rank of Undiluted Group	P-value
Generics/1995	3003	2	**0.0007**
Generics/1994	924	61	0.0660
NASA/1995	1716	67	**0.0390**
NASA/1994	924	401	0.4340

Table 3. Results of permutation tests for team scores.

3.3 Analysis for Perspectives

3.3.1 Analysis Strategy for Perspectives

We were also concerned with the question of whether the perspectives used in the experiment are useful (i.e., reviewers using each perspective contributed a significant share of the total defects detected) and orthogonal (i.e., perspectives did not overlap in terms of the set of defects they helped detect). A full study of correlation between the different perspectives and the types and numbers of errors they uncovered will be the subject of future work, but for now we take a qualitative look at the results for each perspective by examining each perspective's coverage of defects and how perspectives overlap.

3.3.2 Results for Perspectives

We formulate no explicit statistical tests concerning the detection rates of reviewers using each of the perspectives, but present Figures 5a and 5b as an illustration of the defect coverage of each perspective. Results within domains are rather similar; therefore we present the ATM coverage charts as an example from the generic domain and the NASA_A charts as an example from the NASA domain. However, due to the differences between experiment runs for the NASA documents, we do not present a coverage diagram for both runs combined. The numbers within each of the circle slices represent the number of defects found by each of the perspectives intersecting there. So, for example, ATM reviewers using the design perspective in the 1995 experiment found 11 defects in total: two were defects that no other perspective caught, three defects were also found by testers, one defect was also found by users, and five defects were found by at least one person from each of the three perspectives.

ATM Results:

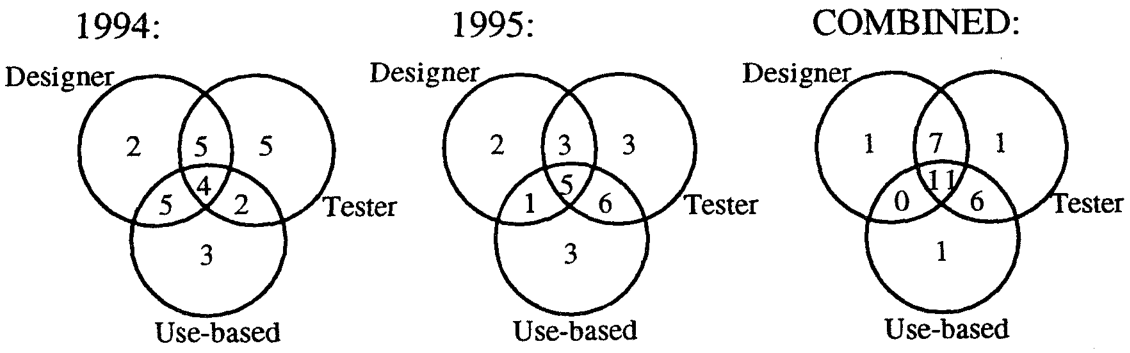
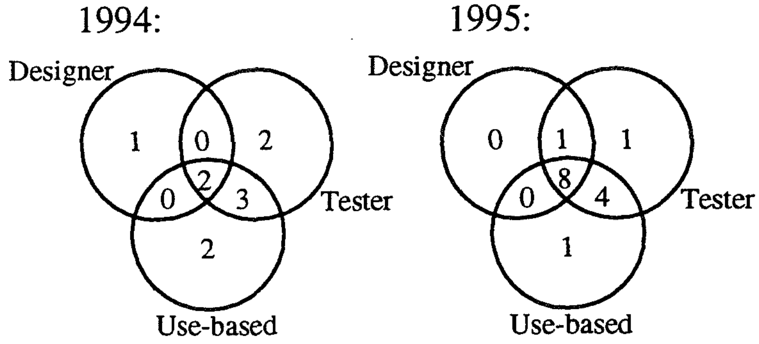


Figure 5a. Defect coverage for the ATM document.

NASA_A Results:


1994:

1995:

Figure 5b. Defect coverage for the NASA_A document.

4. PBR Effectiveness

In the previous section we presented the analysis of the data from a strictly statistical point of view. However, it is necessary to assess the meaning and implications of the analysis to see if we can identify trends in the results that are similar for both runs of the experiment. Such interpretations may also point out areas of weakness in the experiment or in the PBR technique - weaknesses which upon recognition become potential areas for improvement.

4.1. Individual Effectiveness

4.1.1. The 1994 Experiment

The individual defect detection rates were better for the generic documents than for the NASA documents in the 1994 replication, regardless of reading technique, because the generic documents were simpler to read and less complex than the NASA documents. Most subjects pointed to the size and complexity of the NASA documents as potential problem areas. However, there is a difference not only in absolute score, but also in the impact the technique has on detection rate. The improvement of PBR over the usual technique was greater for the generic documents than for the NASA documents. We can think of various reasons for this:

- The perspectives and the questions provided were not aimed specifically at the NASA documents, but based on the general nature of the generic documents.

Thus the technique itself may not be exploited to its full potential for documents within the NASA domain.

- It is possible that the reviewers are more likely to fall back on their usual technique rather than apply the PBR technique when reading documents that they are familiar with. We received anecdotal evidence of this during follow-up interviews. This may be of particular importance in situations where the subjects are under pressure due to time constraints and the complexity of the document.
- The 1994 experiment was carried out in the reviewers' own work environment. This may increase the temptation to fall back to the usual technique when the familiar situation of reading NASA documents arose. The generic documents, on the other hand, would not be likely to stimulate such interaction effects.
- Insufficient training may have been provided since the training sessions only explained how to use the technique on a sample generic document and not on a sample NASA document.

Within each of the two domains, we found that the documents were at the same level of complexity with only minor differences between them. This indicated that our effort of keeping the documents within each domain comparable was successful.

4.1.2. The 1995 Experiment

In the 1995 replication we made some changes to account for some of the problems mentioned above. The NASA documents were modified substantially according to the comments we received from the subjects. We also provided additional training by adding two more sessions aimed at applying the techniques to the NASA documents. The experiment itself was carried out in a classroom environment instead of the work environment. However, even though we saw a substantial rise in the absolute defect detection rates for the NASA documents, the improvement of PBR over the usual technique remained insignificant. Thus our most viable explanation at the moment is that PBR needs to be more carefully tailored to the specific characteristics of the NASA documents and environment to show an improvement similar to what we see in the generic domain. We also got feedback from the subjects that supported this view; several found it tempting to fall back to their usual technique when reading the NASA documents.

For the generic domain, we made only minor changes to the documents and the seeded defects. Thus, we expected the change in defect detection rate to be negligible. However, this appeared not to be the case.

The mean detection rate for the ATM document turned out to remain unchanged, but dropped significantly for the PG document. We have analyzed this carefully, but have not been able to find a plausible explanation as to why this should happen. Changes to the experiment should be expected to have a similar impact on the two documents, so perhaps the changes to the two documents were not as insignificant as we thought.

4.1.3. Combined

Although the changes to the NASA documents were a definite improvement, any effect due to technique is hidden by the much larger difference between the two runs of the experiment. This problem illustrates one of the tradeoffs we had to make when planning the second run. Should we have kept the documents unchanged, thus getting data that may not be completeiy valid, or should we change the documents but get data that would be hard to combine with the data from the initial run? We chose to change the documents, and in retrospect we feel the right decision was made.

We did not have the same problems with the generic documents because they were changed only slightly between the two runs of the experiment. Thus the results indicate a significant improvement of the defect detection rate in the generic domain due to the application of PBR.

4.2. Teams

4.2.1. The 1994 Experiment

The defect detection rates of teams in the 1994 experiment reflected the same trends as the individual rates. For the NASA documents, the defect detection rates were much lower than they were for the generic documents, regardless of reading technique. But even more importantly, the results from the permutation test indicate that there are only random differences between the two techniques in this case. This, together with the defect coverage discussed in section 3.2, counts as evidence that the current perspectives do not work as well with the NASA documents as they do with the generic documents.

4.2.2. The 1995 Experiment

In the 1995 experiment, the team results for the generic documents showed that using PBR resulted in a significant improvement over the usual technique. The reasons for this observed improvement, as compared to the 1994 experiment, may include better training sessions and a less intrusive environment, which in the 1995 experiment was a classroom setting. This environment may have made it easier to concentrate on the experiment and thus to keep the two techniques independent from each other.

For the NASA documents, the results were also better than in 1994. In addition to the possible explanations mentioned for the generic documents, there is the fact that there were substantial changes to the documents. Thus, the results provide more evidence for the 1994 indication that the subjects tend to use their usual technique when reading familiar documents in a familiar work environment, and in particular when under pressure.

4.3. Threats to Validity

The threats to internal validity discussed in section 2 may have an impact on the results of the experiment. Thus, at this point it may be interesting to see whether the potential impact and the results agree. Below we discuss the threats that we find most important:

- **History:** One problem with our experiment is that it does not allow history effects to be separated from the change in technique. Since there was one day between the two days of the experiment, some of the improvement that appears due to technique may be attributed to other events that took place between the tests. We do not consider this effect to be very significant, but we cannot completely ignore it.
- **Maturation:** We may assume the results obtained in the afternoon to be worse than the results from the morning session because the subjects may get tired and bored. Since the ordering of documents and domains was different for the two days, the differences between the two days may be disturbed by maturation effects. Looking at the design of the experiment, we see that an improvement from the first to the second day would be amplified for the generic documents, while it would be lessened for the NASA documents. Based on the results from the experiment, we see that this effect seems plausible.

- **Testing:** This may result in an improvement in defect detection rate due to learning the techniques, becoming familiar with the documents, becoming used to the experimental environment and the tests. This effect may amplify the effects of the historical events and thus be part of the reason for improvement that has previously been considered a result of change in technique. Testing effects may counteract maturation effects within each day.
- **Reactive effects:** The change of experimental environment between the experiment runs may have made it easier to concentrate on the techniques and tests to be done, thus separating the techniques better for the second run of the experiment.

We cannot say anything conclusive about the impact of threats to validity. However, we feel that we have taken them into account as carefully as possible, given the nature of the problem and our experimental design.

Since the two runs of this experiment have been done in close cooperation with the NASA SEL environment, it seems natural to conclude this section with a discussion of the extent to which the results can be generalized to a NASA SEL context. This kind of generalization involves less of a change in context than is the case for an arbitrary organization; in particular the differences in populations can be ignored since the population for the experiments is in fact all of the NASA SEL developers.

Clearly, the results for the generic documents cannot be generalized to the NASA documents due to the difference in nature between the two sets of documents. The results for the NASA documents, on the other hand, may be valid since we used parts of *real* NASA documents. Finally, there is a potential threat to validity in the choice of experimental environment. In 1994, the experiment was carried out in the subjects' own environment, and thus would be valid also in a real setting. We cannot assume the same for the 1995 results since this run was done in a classroom situation.

5. Observations on Experimental Design

We have encountered problems in the two runs of the experiment which we have previously discussed. However, some of these problems are of a general nature and may be relevant in other experimental situations.

- *What is a good design for the experiment under investigation, given the constraints?*

There appears to be no easy answer to this question. Each design will be a result of a number of tradeoffs, and it is not always possible to know how the decisions will influence the data. A good design can have various interpretations based on what are considered the goals for the experiment. One option is to use different designs involving different threats to validity and study the results as a whole.

- *What is the optimal sample size? Small samples lead to problems in the statistical analysis while large samples represent major expenses for the organization providing the subjects.*

  Organizations generally have limits for the amount of subjects they are willing to part with for an experiment, so the cost concerns are handled by the organizations themselves. A small sample size requires us to be careful in the design in order to get as many useful data points as possible. For this experiment, an example of such a tradeoff is that we chose to neglect learning effects in order to avoid spending subjects on control groups. This gave us more data points to be used in analyzing the difference between the two techniques, but at the same time we remained uncertain as far as the threat to internal validity caused by learning effects is concerned.

- *We need to adjust to various constraints - how far can we go before the value of the experiment decreases to a level where it is not worthwhile?*

  Our problem as experimenters is to maintain a certain level of validity while still generating sufficient interest for an organization to allow us to conduct the experiment. From an organization's point of view, an experiment should be closely tied to their own environment to see if the suggested improvement works with minimal effort in terms of environmental changes. From an experimental point of view, however, we are interested in a controlled environment where disturbing interaction effects are negligible.

- *To what extent can experimental aspects such as design, instrumentation and environment be changed when the experiment still is to be considered a replication?*

  One requirement for being considered a replication is that the main hypotheses are the same. Changes in design and instrumentation, in particular to overcome threats to validity, should also be considered "legal". However, one situation we should avoid is making substantial changes to the design based on the *results* from a previous experiment. This will introduce dependencies between the experiments that are highly undesirable from a statistical point of view.

For this experiment in particular, there are various problems that we need to study more carefully. The threats to validity should be carefully examined; in particular we feel the testing effects to be crucial. An experiment with a control group could be one way of estimating what the importance of these effects really are. We may also consider a more careful analysis of the NASA documents and environment in order to refine PBR to these particular needs. The results indicate that the choice of perspectives and associated scenarios do not match the needs of the NASA domain.

A more fundamental problem that should be considered is to what extent the proposed technique actually is followed. This problem with process conformance is relevant in experiments, but also in software development where deviations from the process to be followed may lead to wrong interpretation of measures obtained. For experiments, one problem is that the mere action of controlling or measuring conformance may have an impact on how well the techniques work, thus decreasing the external validity.

Conformance is relevant in this experiment because there seems to be a difference that corresponds to experience level. Subjects with less experience seem to follow PBR more closely ("It really helps to have a perspective because it focuses my questions. I get confused trying to wear all the hats!"), while people with more experience were more likely to fall back to their usual technique ("I reverted to what I normally do.").

There are numerous alternative directions for the continuation of this research. For further experimentation within NASA's SEL it seems to be necessary to tailor PBR to more closely match the particular needs of that domain. A possible way of further experimentation would be to do a case-study of a NASA SEL project to obtain more qualitative data.

We may also consider replication of the generic part of the experiment in other environments, perhaps even in other countries where differences in language and culture may cause effects that can be interesting targets for further investigation. These replications can take the form of controlled experiments with students, controlled experiments with subjects from the industry using their usual technique for comparison, or case studies in industrial projects.

One challenging goal of a continued series of experiments will be to assess the impact that the threats to validity have. Since it is often hard to design the experiment in a way that controls for most of the threats, a possibility would be to concentrate on certain threats in each replication to assess their impact on the results. For example, one replication may use control groups to measure the effect of repeated tests, while another replication may test explicitly for maturation effects. However, we need to keep the replications under control as far as threats to *external* validity are concerned, since we need to assume that the effects we observe in one replication will also occur in the others.

Acknowledgements

This research was sponsored in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland. We would also like to thank the members of the Experimental Software Engineering Group at the University of Maryland for their valuable comments to this paper.

References

(Campbell, 1963) Campbell, Donald T. and Stanley, Julian C. 1963. *Experimental and Quasi-Experimental Designs for Research* . Boston, MA: Houghton Mifflin Company.

(Edington 1987) Edington, Eugene S. 1987. *Randomization Tests.* New York, NY: Marcel Dekker Inc.

(Fagan, 1976) Fagan, M. E. 1976. *Design and code inspections to reduce errors in program development.* IBM Systems Journal, 15(3):182-211.

(Hatcher, 1994) Hatcher, Larry and Stepanski, Edward J. 1994. *A Step-by-Step Approach to Using the SAS® System for Univariate and Multivariate Statistics.* Cary, NC: SAS Institute Inc.[2]

(Heninger, 1980) Heninger, Kathryn L. 1985 *Specifying Software Requirements for Complex Systems: New Techniques and Their Application.* IEEE Transaction on Software Engineering, SE-6(1):2-13

(Linger, 1979) Linger, R. C., Mills H. D. and Witt, B. I. 1979. *Structured Programming: Theory and Practice.* In The Systems Programming Series. Addison Wesley.

(Parnas, 1985) Parnas, Dave L. and Weiss, David M. 1985. *Active design reviews: principles and practices.* In Proceedings of the 8th International Conference on Software Engineering, p.215-222.

(Porter, 1995) Porter, Adam A., Votta, Lawrence G. Jr. and Basili, Victor R. *Comparing Detection Methods For Software Requirements Inspections: A Replicated Experiment.* IEEE Transactions on Software Engineering, June 1995.

(SAS, 1989) SAS Institute Inc. 1989. *JMP® User's Guide.* Cary, NC: SAS Institute Inc.[3]

(SEL, 1992) Software Engineering Laboratory Series. 1992. *Recommended Approach to Software Development, Revision 3*, SEL-81-305, p. 41-62.

(Votta, 1993) Votta, Lawrence G. Jr. 1993 *Does every inspection need a meeting?* In Proceedings of ACM SIGSOFT '93 Symposium on Foundations of Software Engineering. Association of Computing Machinery, December 1993.

A. Sample Requirements

Below is a sample requirement from the ATM document which tells what is expected when the bank computer gets a request from the ATM to verify an account:

Functional requirement 1

Description: The bank computer checks if the bank code is valid. A bank code is valid if the cash card was issued by the bank.

Input: Request from the ATM to verify card (Serial number and password)

Processing: Check if the cash card was issued by the bank.

Output: Valid or invalid bank code.

We also include a sample requirement from one of the NASA documents in order to give a picture of the difference in nature between the two domains. Below is the process step for calculating adjusted measurement times:

Calculate Adjusted Measurement Times: Process

1. Compute the adjusted Sun angle time from the new packet by

$$t_{s,adj} = t_s + t_{s,bias}$$

2. Compute the adjusted MTA measurement time from the new packet by

$$t_{T,adj} = t_T + t_{T,bias}$$

3. Compute the adjusted nadir angle time from the new packet.

a. Select the most recent Earth_in crossing time that occurs before the Earth_in crossing time of the new packet. Note that the Earth_in crossing time may be from a previous packet. Check that the times are part of the same spin period by

$$t_{e-in} - t_{e-out} < E_{\max} T_{spin,user}$$

b. If the Earth_in and Earth_out crossing times are part of the same spin period, compute the adjusted nadir angle time by

$$t_{e-adj} = \frac{t_{e-in} + t_{e-out}}{2} + t_{e,bias}$$

4. Add the new packet adjusted times, measurements, and quality flags into the first buffer position, shifting the remainder of the buffer appropriately.

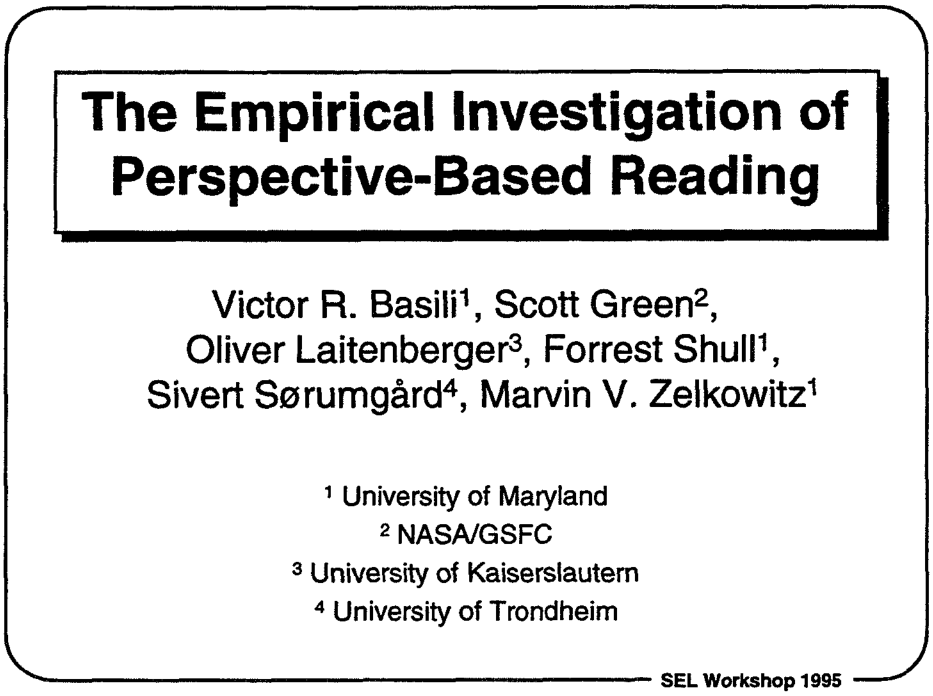
5. The Nth buffer position indicates the current measurements, observation times, and quality flags, to be used in the remaining Adjust Processed Data section. If the Nth buffer does not contain all of the adjusted times ($t_{s,adj}, t_{b,adj}, t_{T,adj}$, and $t_{e,adj}$), set the corresponding time quality flags to indicate invalid data.

Footnotes

[1] ISERN is the International Software Engineering Research Network whose goal is to support experimental research and the replication of experiments.

[2] SAS® is the registered trademark of SAS Institute Inc.

[3] JMP® is a trademark of SAS Institute Inc.

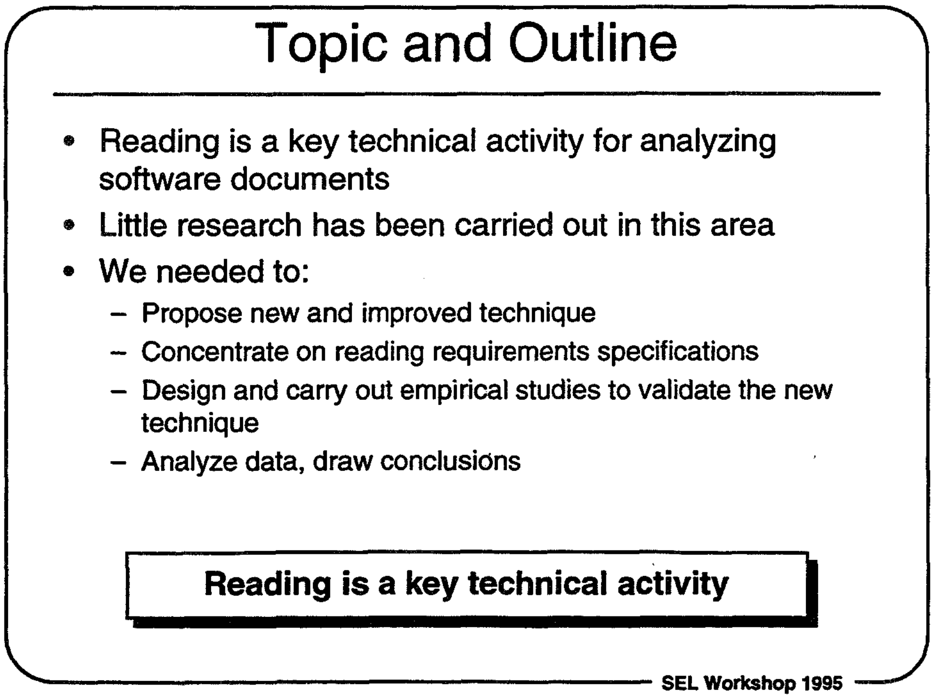


The Empirical Investigation of Perspective-Based Reading

Victor R. Basili[1], Scott Green[2],
Oliver Laitenberger[3], Forrest Shull[1],
Sivert Sørumgård[4], Marvin V. Zelkowitz[1]

[1] University of Maryland
[2] NASA/GSFC
[3] University of Kaiserslautern
[4] University of Trondheim

SEL Workshop 1995

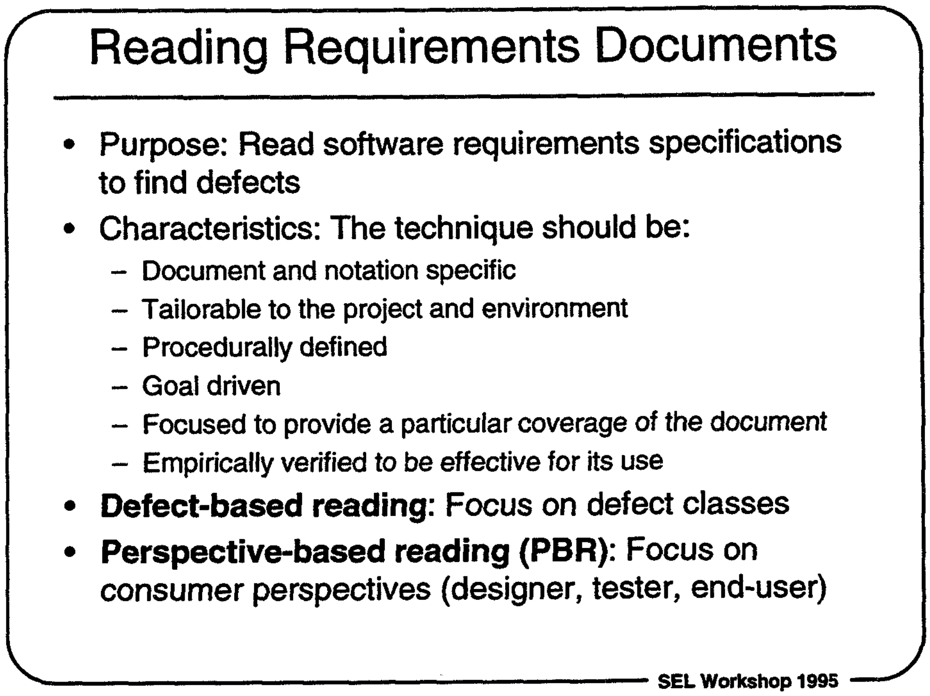


Topic and Outline

- Reading is a key technical activity for analyzing software documents
- Little research has been carried out in this area
- We needed to:
  - Propose new and improved technique
  - Concentrate on reading requirements specifications
  - Design and carry out empirical studies to validate the new technique
  - Analyze data, draw conclusions

Reading is a key technical activity

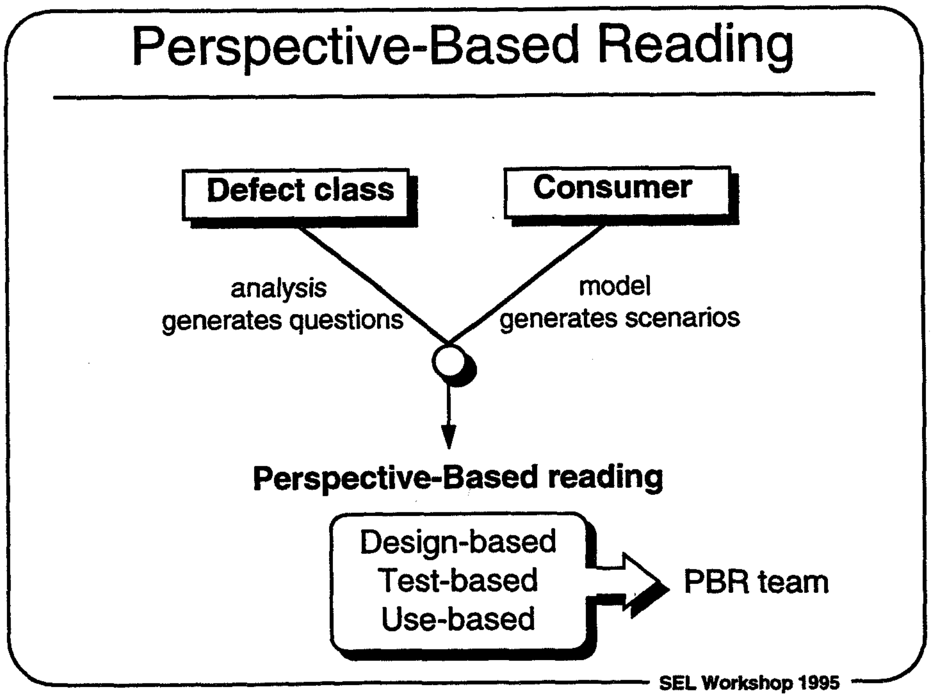
SEL Workshop 1995



Reading Requirements Documents

- Purpose: Read software requirements specifications to find defects
- Characteristics: The technique should be:
  - Document and notation specific
  - Tailorable to the project and environment
  - Procedurally defined
  - Goal driven
  - Focused to provide a particular coverage of the document
  - Empirically verified to be effective for its use
- **Defect-based reading**: Focus on defect classes
- **Perspective-based reading (PBR)**: Focus on consumer perspectives (designer, tester, end-user)

SEL Workshop 1995



Perspective-Based Reading

SEL Workshop 1995

PBR Example

- Test-based reading (excerpt):
  For each requirement/functional specification, generate a test or set of tests that allow you to ensure that an implementation of the system satisfies the requirement/functional specification. Use your standard test approach and technique, and incorporate test criteria in the test suite. In doing so, ask yourself the following questions for each test:

1. Do you have all the information necessary to identify the item being tested and the test criteria? Can you generate a reasonable test case for each item based upon the criteria? Can you be sure that the tests generated will yield the correct values in the correct units?

2. Can you be sure that the tests generated will yield the correct values in the correct units?

... etc.

Questions for each perspective

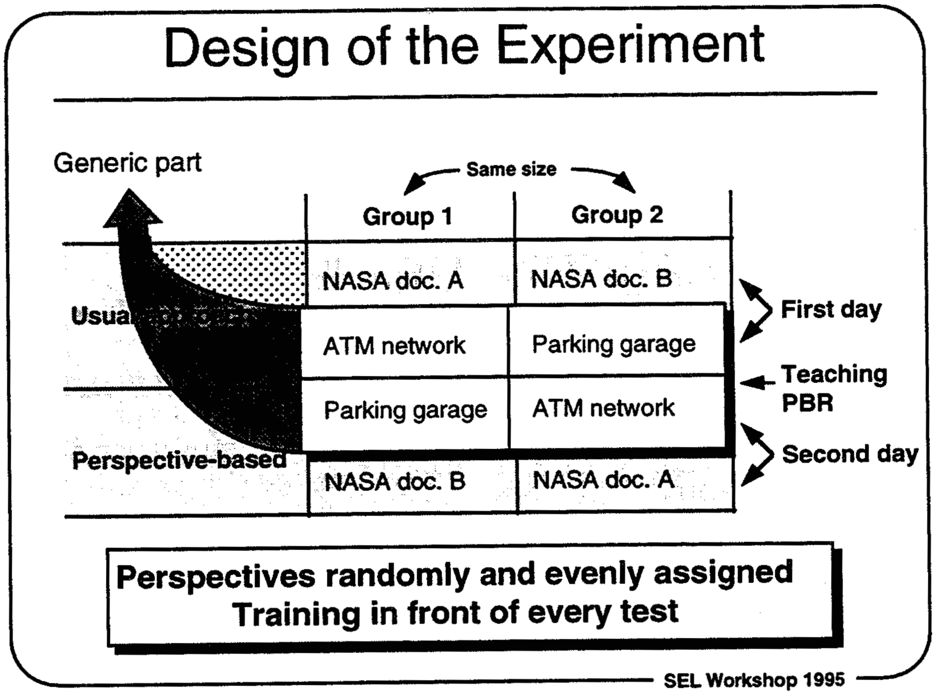
SEL Workshop 1995

PBR Experiment

Goal:
Analyze perspective-based reading in order to evaluate it with respect to the individual and team effect on defect detection effectiveness of NASA's current reading technique from the viewpoint of quality assurance

- Environment: NASA/CSC SEL
  - Two structured text generic documents (ATM, PG), two NASA functional spec. (ground support sub-systems)
  - All documents seeded with known sets of defects
  - Metric: Defect detection rate as % of defects
  - Two hour time limit, actual time measured but not used in analysis
  - Carried out twice: November 1994 and June 1995
  - 25 subjects in total

SEL Workshop 1995

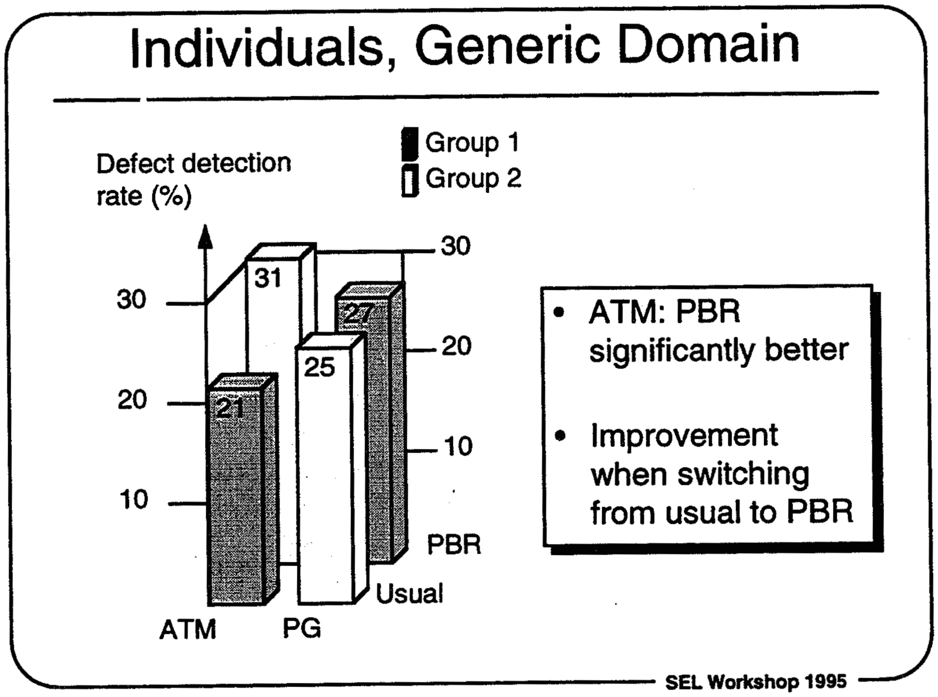


Design of the Experiment

	Group 1	Group 2	
Generic part	NASA doc. A	NASA doc. B	First day
Usual approach	ATM network	Parking garage	First day
			Teaching PBR
Perspective-based	Parking garage	ATM network	Second day
Generic part	NASA doc. B	NASA doc. A	Second day

Perspectives randomly and evenly assigned
Training in front of every test

SEL Workshop 1995

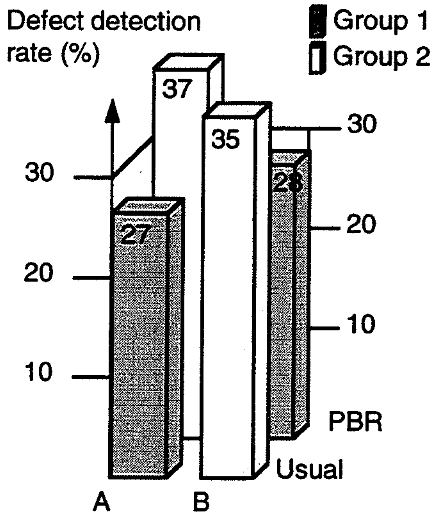


Individuals, Generic Domain

- ATM: PBR significantly better
- Improvement when switching from usual to PBR

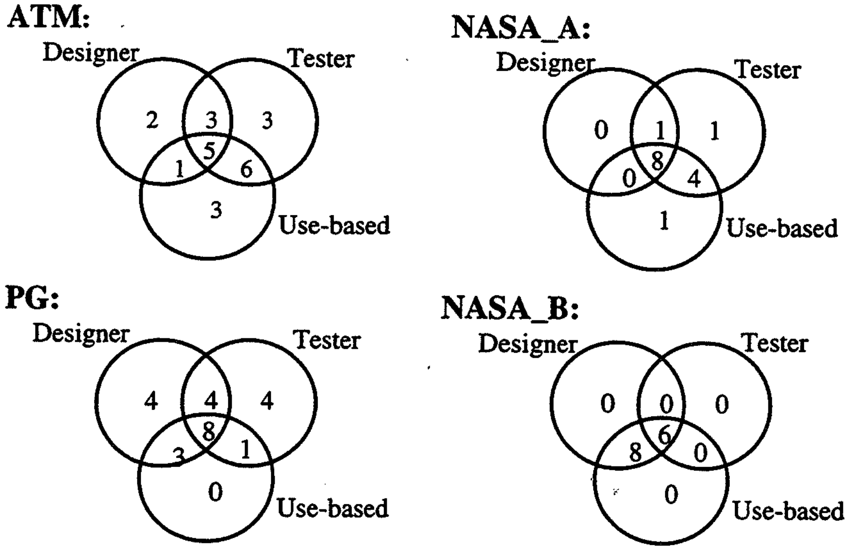
SEL Workshop 1995

Individuals, NASA Domain



- PBR not significantly better for any document
- No significant change when switching from usual to PBR

Perspective Coverage (1995)

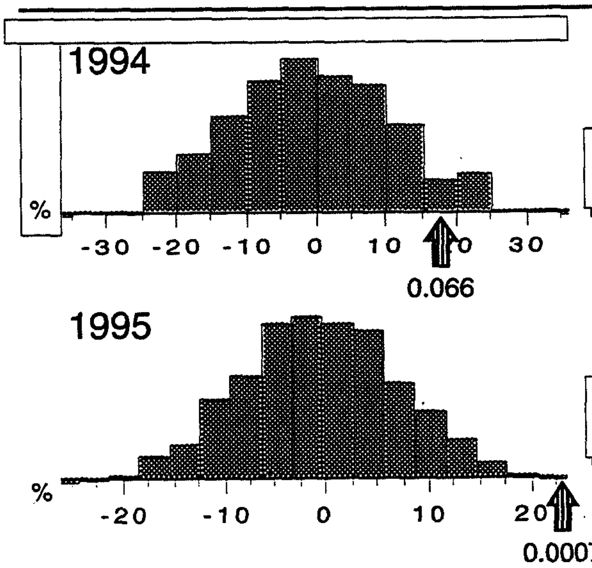


Perspective Coverage (cont.)

- Were perspectives
  - **useful** (did they catch a *significant number* of defects)?
  - **orthogonal** (did they catch *different* defects)?

	Useful	Orthogonal
Generic	**Yes**	**Yes**
NASA	**Maybe**	**No**

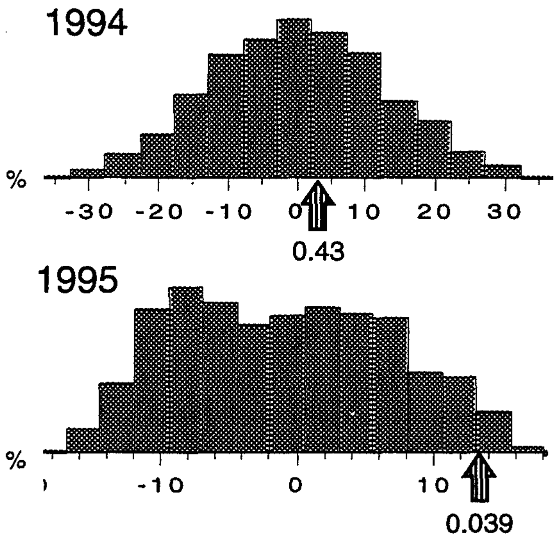
Teams, Generic Domain



Almost significant

Very significant

Teams, NASA Domain



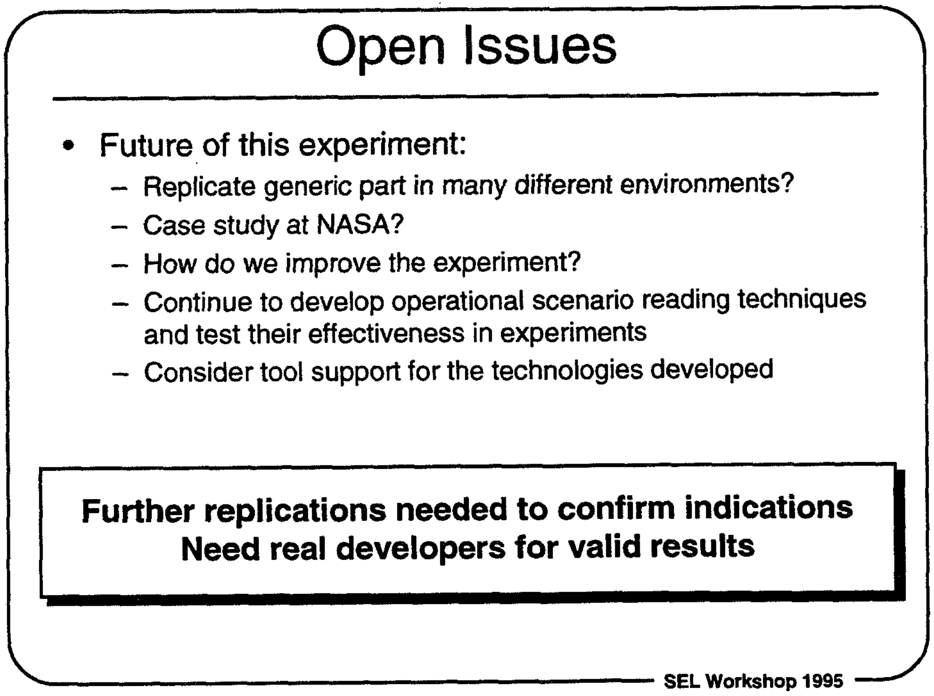
No effect

Significant

PBR Conclusions

- Observations:
  - PBR is most successful in the generic domain
  - PBR is not sufficiently tailored to the NASA environment in terms of document contents, notation and perspectives
  - Relative benefit of PBR seems to be higher for teams
- Possible explanation:
  - Subjects seem to fall back to their familiar technique when reading a familiar document

Basic idea behind PBR seems to work
Tailor PBR to domain to achieve full potential



Open Issues

- Future of this experiment:
  - Replicate generic part in many different environments?
  - Case study at NASA?
  - How do we improve the experiment?
  - Continue to develop operational scenario reading techniques and test their effectiveness in experiments
  - Consider tool support for the technologies developed

Further replications needed to confirm indications
Need real developers for valid results

SEL Workshop 1995

Porting Experience Factory Concepts to New Environments

S3-61

[illegible]

360687

22P.

Frank McGarry
Computer Sciences Corporation
7700 Hubble Drive
Lanham-Seabrook, MD 20706
(301) 794-2450
fmcgarry@csc.com

As our approach to software process improvement has matured, so have the defined paradigms for characterizing the improvement steps. The Experience Factory (EF) is a concept comprising both structure and activities that was first formally defined by Basili (Reference 1) in 1989. The model evolved from specific experiences at NASA/Goddard's Software Engineering Laboratory, which was established in 1976 and has continued to be an operational Experience Factory for nearly 20 years (Reference 2). Other environments have also attempted to incorporate the key elements of the EF concept for the purpose of implementing a goal-driven software improvement program.

This paper captures the experiences of a sample set of organizations that have attempted to apply the EF concepts. The paper addresses cost, timelines, impediments to success, and lessons learned, as reported by those organizations that volunteered such information on their experiences. Approximately eight organizations had some level of information available on at least one or two key elements of their EF implementation efforts. The organizations ranged in size from those with 40 or 50 software engineers to those with over 5000 employees. This paper is a synthesis of the information provided by these groups.

Introduction

For over 20 years, the Software Engineering Laboratory (SEL) at NASA/Goddard has been carrying out studies in software process improvement toward the goal of generating improved software within this one NASA domain. The concepts used in this improvement program were formalized by Basili (Reference 1) in 1989 and are called the "Experience Factory" (EF). The EF comprises both an organizational structure and a set of activities focused on continual improvement within an organization as measured against the goals of that organization. The concepts have been replicated in other software organizations besides NASA/Goddard, and this paper reports on the experiences resulting from attempts to apply the EF approach in these broader and varying software domains.

The EF differs from models such as the Capability Maturity Model (CMM) (Reference 3). One significant difference is that the EF presents a paradigm for continuous change as opposed to a model for rating a process against some benchmark. It entails a set of activities and a structure that focus on process change and improvement as opposed to focusing on process itself. Although most software development organizations have some type of improvement program in place, the concept of EF is different from many of the common improvement concepts and is not as widely applied as the CMM-driven approach. In this paper, only organizations that are specifically and explicitly attempting to apply these concepts were used as a source.

The information captured in this paper represents experiences of the author and colleagues in initiating improvement programs at different sites. Each of the efforts had the goal of applying the EF approach to the new program. Most of the reported experiences are based on subjective data as opposed to specific quantified information relating to the efforts. This summary may provide some guidance for organizations setting out to adopt an overall software process improvement program using concepts captured in the EF.

Terminology

Since one of the distinguishing characteristics of the EF approach is the concern for change in both product and process as opposed to process only, it is necessary to clarify some of the relevant terms.

Process pertains to how the software product is generated. It includes the steps, methods, techniques, and organizational structure used to carry out the task of software development and maintenance. Essentially, it consists of all the attributes that would be reviewed by CMM baselining activity. Samples include review activities, testing approach, design approach, inspections used, quality assurance (QA) techniques, and life cycle applied.

Product refers to the end items generated as part of the development or maintenance activity. It includes software and the associated documentation.

Process and product measures refer to the attributes that can be determined from the steps used (process) and the end items generated (product). *Process measures* include such items as percentage of time spent in design, number of inspections performed, and number of tests executed. *Product measures* include such items as code size, number of pages of documentation, number of delivered defects, cost of the product, productivity, or number of defects per unit of size.

Some measures, such as language used, may be considered both a process measure and a product measure. The fact that some measures can be either process or product measures does not have any impact on our application.

Characteristics of the EF

The EF organization implies two separate elements. The first is the software production organization, which develops and maintains software and is the source of experience. The second is the Experience Factory, a separate organizational unit that supports reuse of experience and collective learning. The EF element is responsible for developing, updating, and delivering experience packages to the software organization. References 1, 2, and 4 present a full discussion of the many attributes of the Experience Factory. The four most prominent attributes that distinguish the EF from other concepts are discussed below.

Concept of experimental software engineering

In the EF, each set of experiences within an organization is leveraged to add knowledge and refinement to the core competencies of the environment. Each software experience (development project, maintenance effort, etc.) is captured as part of the continually increasing depth of capabilities. Some of these experiences are captured from pilot projects and some from controlled experiments, but most are derived from routine software activities. Every software project is considered an experiment whereby new knowledge is acquired during and at the completion of the effort.

Goal-driven change

As with any improvement concept, the goal of the EF is to provide guidance toward better software, with "better" defined by the organization. Improvement is not measured by the adoption of more mature processes but is measured against the products of the organization, whose goals are established a priori. For a typical goal such as decreasing defects by 50%, the measure of success would be based only on that goal, not on whether or not appropriate defect-prevention processes were adopted. Obviously, some change would have to be made to process or technology to target the product improvement goal.

Separation of concerns

The EF concept emphasizes that the software production organization cannot be burdened with responsibility for the overall execution of the improvement program; resources must be allocated to the EF element itself. The production staff focuses on producing software on time and within budget and is kept separate from the EF element, which handles the analysis and information repository. This separation of concerns allows each of the elements to focus on the tasks they are most competent to carry out and does not divert resources from the project staff itself. One of the oldest existing Experience Factories is the Software Engineering Laboratory, which is described in Reference 2 and whose structure is depicted in Figure 1.

Measurement

Software measurement is critical to the EF concept to verify the need for change, identify the effects of the changed process, and continually build the engineering concepts of the environment. The Experience Factory cannot exist without adopting, using, and relying on software measurement. It is a necessity from the very first step in establishing the overall EF concept.

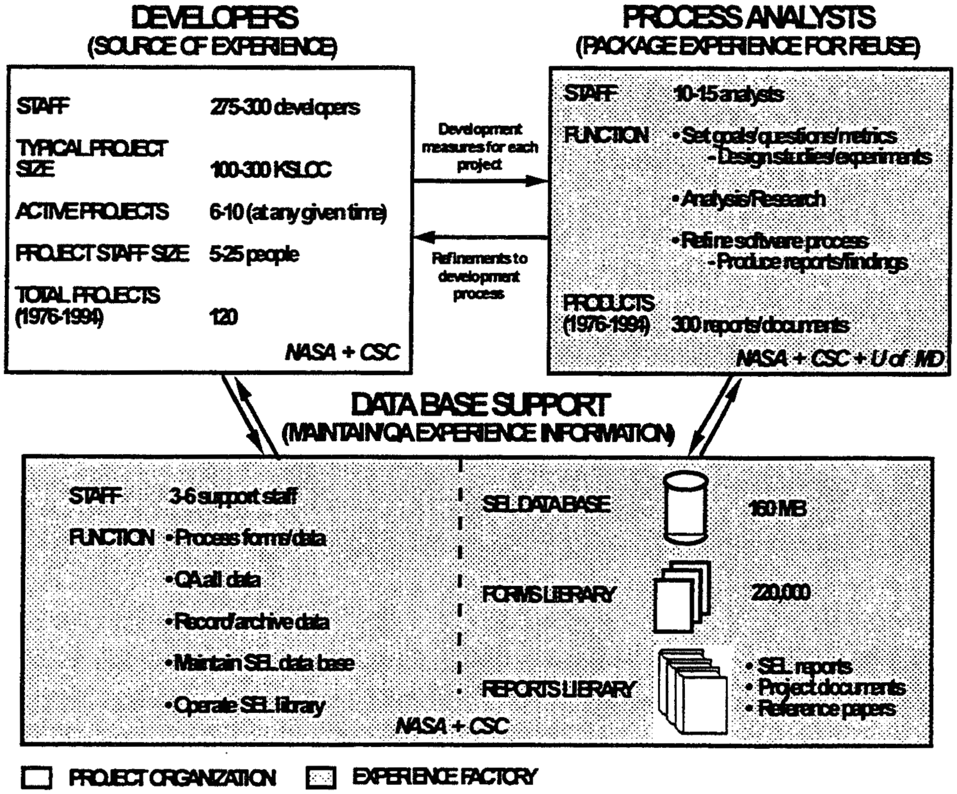


Figure 1. The Software Engineering Laboratory Experience Factory

Information Sources

To generalize the impacts and effects of the EF concepts for this paper, information from several sources was reviewed. The information from the SEL is the most complete and extensive and provides the baseline of the observations made. Additional sources include the following:

- Additional NASA activities
  Center-wide activities at Goddard and at Jet Propulsion Laboratory
  Agency wide efforts (through Code Q)
  Several local programs such as the SEAL at NASA/Langley
- Computer Sciences Corporation
  Flight Dynamics Technology Group (of the SEAS program)
  SEAS-wide activities
- National Security Agency (NSA)

- Hughes Information Sciences Division
  EOS Program
- European companies working with the University of Kaiserslautern,
  including Robert Bosch, Daimler Benz, Nokia, and Ericsson

Each of these organizations provided some experience data and information about establishing improvement programs based on the EF concepts. None of the organizations had complete information relating to this study, so partial data was applied. There was no attempt to apply rigorous qualitative analysis of the information since this study was merely attempting to report experiences in a general fashion.

The basic information sought from each organization included

- Cost of establishing the structure
- Key initial products generated
- Timelines for generating major products
- Key lessons learned
- Identified payoff
- Most disappointing and most successful program elements after initial year of activity

Key Lessons From the Broadened Experiences

From reviewing and comparing the information provided by the organizations, eight prominent points emerged that seemed to summarize the experiences of these groups. Some had to do with cost, others with timelines, and most were associated with the subjective experience of applying the concept to production organizations. The eight points are addressed below.

Lesson 1. There are four specific, initial products that are valuable in establishing an EF.

Only three of the organizations (besides the SEL) had completed all four of the key products listed below. However, all of the organizations indicated either that they intended to complete the product or that they felt it should have been completed during the first year of operation. The four products are as follows:

- **Improvement Plan**— Major elements of this plan include the organizational goals, concepts of improvement (EF), approach, and target schedules for implementing the program and making it operational.
- **Organization Baseline**—This product was identified as one of the most important and distinguishing products of the EF implementation. The baseline captures both process and product information and provides the benchmark for identifying change and improvement. Each contributing organization indicated that capturing the starting point of the ongoing process (e.g., by completing a Software Process Assessment) should be complemented with the initial product data and information. Critical elements of the baseline are shown in Table 1. The baseline must not judge the

validity or adequacy of the organization's process and products, but it should capture the key process activities being used. Otherwise, it cannot be determined if change is being applied.

Table 1. Critical Elements of the Organization Baseline

Process Elements	Product Elements
Key elements of written process	Size characteristics
Major components of process in use	Cycle time
Role of management - Quality assurance (QA) - Configuration management (CM) - Project structure	Cost - Total development - Maintenance rate - By function (QA/CM/testing)
Improvement activities	Defects (number and type)
Perceptions of developers	Defects by test phase

- **Software Process**—Most organizations typically have a set of software policies and standards that describe the expected process for software efforts. To identify change and track evolution, it is important to ensure that the identified policies and standards adequately describe the software process being applied within the organization. If significant differences exist between the written process and that which is being applied, it is vital to capture the key attributes of the process that is inherent in the organization.

- **Measurement Program**—Inherent in all the concepts of the EF is that of measurement. It is a major element of baselining, setting goals, determining change and experimenting. Therefore, it must be part of any effort to establish an EF. The required measures are defined by the goals of the organization as well as by specific projects, but the overall operation of the measurement program must be established right from the start.

Lesson 2. Start-up costs are insensitive to domain size.

The size and scope of the programs planned by the participating organizations varied tremendously. However, the resources expended in the basic activity during the first 1 to 2 years was relatively consistent; at least, it did not seem directly related to organization size. It may be tied directly to available budgets or to similar limitations, but the reported expenditure on the four key startup products was relatively consistent. The organizations ranged in size from approximately 50 software personnel to well over several thousand. For the four organizations that provided data, the effort ranged from 4% to 7%. Figure 2 shows these typical costs.

For the four organizations that provided data, the improvement plan cost ranged from 3 to 6 staff-months of effort; the baseline cost ranged from 8 staff-months to 14 staff-months; the effort required to capture the ongoing process typically ran about 15 staff-months; and the effort to establish the measurement program was approximately 1 staff year. To establish the measurement program, the organizations had to define the measures, produce the collection mechanism, and establish the archiving and analysis process.

Figure 3 shows the associated timelines for the key products produced during the first two years. It also shows a timeline for initiating experiments or studies; two of the surveyed organization reported that they had initiated a few studies, but indicated that it takes a long time to get them started.

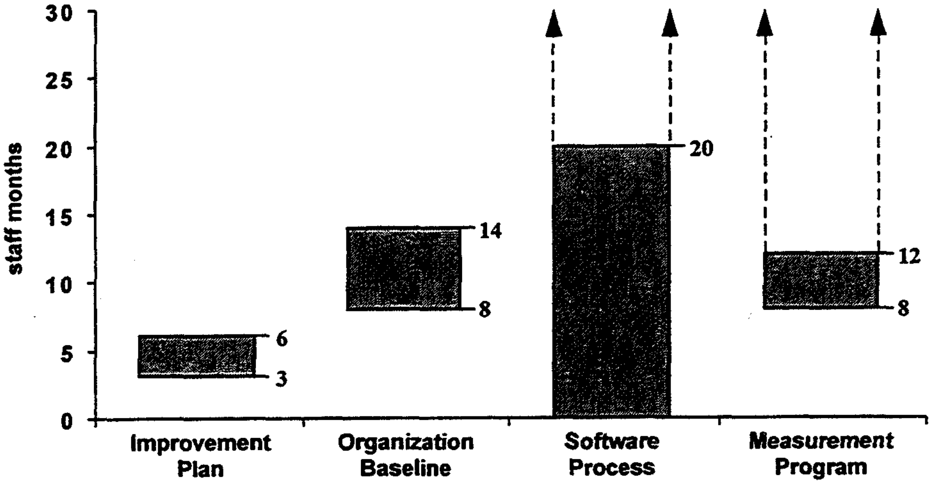


Figure 2. Experience Factory Startup Costs

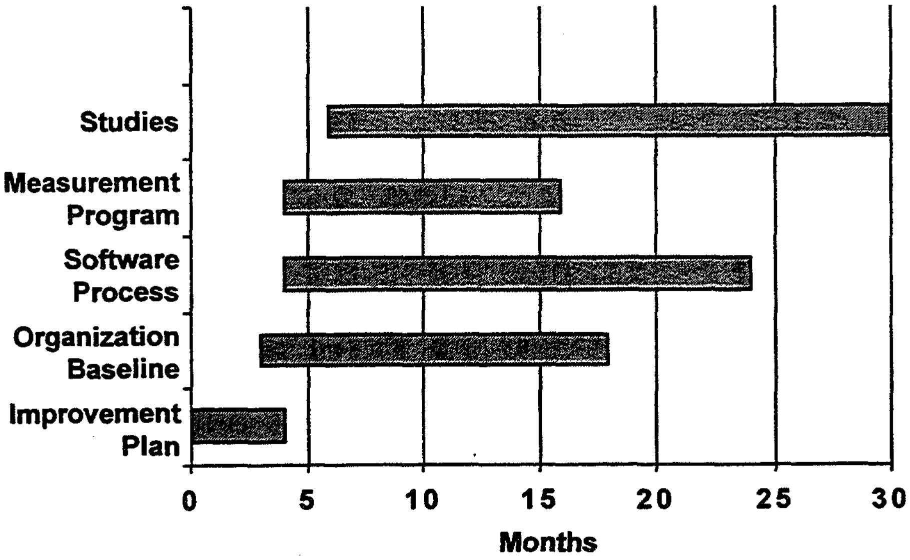


Figure 3. Experience Factory Startup Timeline

Lesson 3. Successful operation is enhanced by the organizational structure.

The organizations reviewed indicated that one of the driving elements determining success is the overall structure within the software organization. The following points derived from their experiences in initiating successful programs are consistent with the original concepts of the EF itself.

- **Separation of concerns**—When the software organization structures the improvement program to focus on two completely separate, yet equally important functions, there is a higher probability of success. The software development staff can focus on producing and maintaining good software. The EF staff can focus on analyzing needs and processes to produce continually improving methods and techniques for the development organization's use.
- **Partnership with research elements**—Several of the more successful EF organizations have relied on research partnerships with universities. The advantage of such a partnership is the continual access to researchers who are interested in probing into methods and technologies as applied to production problems. The university has access to resources such as enthusiastic (and inexpensive) graduate students, and to a network of related research institutions such as other universities. The local organization is then better able to focus on the development, packaging, and overall analysis.
- **Rotational assignments to broaden experience**—With the development and EF elements separated, there is concern that individuals no longer have the opportunity to carry out both functions. Most EF organizations are now targeting to .rotate individuals through both elements to provide broader experience and to provide opportunity for career growth.

Lesson 4. The cost of operating an EF ranges from 4% to 10% of the software budget.

Figure 4 shows the costs of operating an EF based on the data provided by organizations who had implemented the EF as the driving concept of an improvement program. For organizations of up to approximately 400 people, the reported cost of the program was approximately 4% to 6% (the SEL was approximately 8%), divided into three categories:

- Overhead to projects—Providing measurement data, attending meetings and training, participating in briefings, etc. Less than 2% (in fact, most places could not measure this overhead because of the insignificant size).
- Measurement data processing—Collecting measures, doing QA, archiving the data, establishing the information repository, and generating basic reports. This function averages about 2% of the software cost.
- Analysis—Designing studies; producing reports; carrying out the analysis; developing new processes, standards, and policies; providing training. This is the

largest cost and averages approximately 4% to 7% (the SEL has the largest cost here, averaging about 7%).

For organizations of a much larger size, the total effort expended is larger, but the percentages of size are much smaller. The values estimated for large organizations (over 500 to 3000 staff) were as follows:

- Overhead to projects—Less than 2% (difficult to measure)
- Measurement data processing—Approximately 1% (or less)
- Analysis—Averages 3%

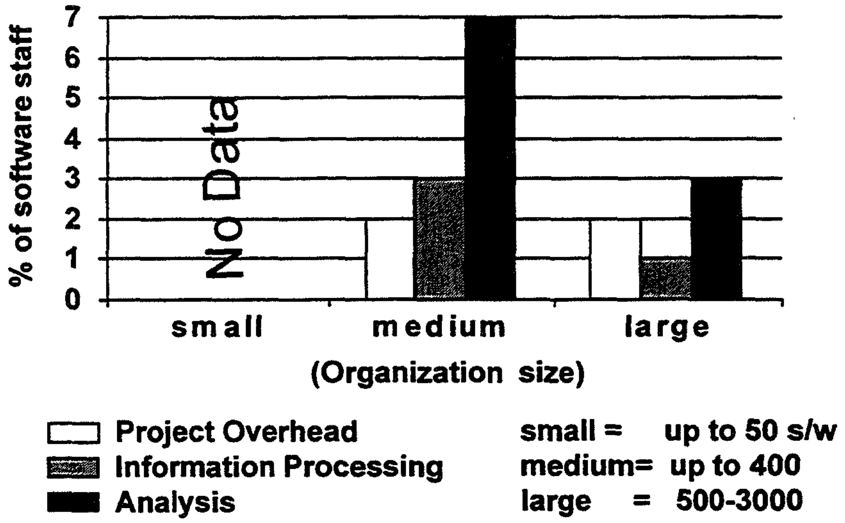


Figure 4. Experience Factory Operating Costs

Lesson 5. Process assessment models are useful as a tool but can be distracting as the goal.

Process assessment models exist and must be used for various business reasons. However, organizations can run into difficulties attempting to apply the EF, while attempting to use models such as the CMM or ISO-9001 as improvement goals. These models can provide tremendous benefits to software organizations who want to institute improvement programs when they are used as tools. Such tools can help identify specific characteristics of software processes in use and can help in producing needed profiles. These tools are detrimental when they are used to completely define the improvement program goals. If the product characteristics are not used as the primary driving force for change, there is no way to determine on a continual basis whether goals are being reached (unless the complete goal is to adopt some standard process).

There is the danger that when change is adopted and a high process model rating is achieved, a false sense of achievement may be acquired. The goal must be to ensure that the specified goals of the organization are being met, that these parameters are continually tracked, and that associated processes are prioritized.

Lesson 6. The ability to measure 'process' has been disappointing.

One of the assumptions of any improvement program is that an improved set of processes will produce an improved product. The goal is to identify and infuse the most beneficial set of process activities so that the end goals of product improvement can be attained.

Information from the organizations reporting on the EF improvement efforts and from other sources indicates only limited success in their attempts to quantify and qualify software processes. Such parameters as quality of tools, maturity of process, quality of inspections, level of structured techniques, and formalisms of reviews are all attempts at measuring process activities, and many of these are required in common cost models. Successes have been reported in counting attributes of processes, such as number of inspections held and the length of time spent on inspections, or in testing. However, only limited success has been reported at measuring the numerous process activities described in many of the models in use today (e.g., CMM, ISO-9001, even basic cost models).

Lesson 7. Value of product-driven process improvement (AKA Experience Factory) has not been widely accepted by the software community.

In gathering information for this paper, it became apparent that the overall concept of letting process change be driven by organizational goals and by specific product measures was not broadly applied. In addition to the information provided by the eight organizations participating in this study, relevant literature and other improvement programs were reviewed to determine how common the basic approaches of the EF may be. The number of readily available examples of environments applying these concepts—even of conducting experimental software engineering—was very limited.

In reviewing some of the relevant articles addressing this point (e.g., Reference 5), it seems there are multiple reasons why the concept of experimental software engineering has not matured or at least has not been commonly adopted. Some of the more obvious reasons include cost, difficulty, and lack of specific guidance on approach.

Improvement programs are on extremely limited budgets, and it requires a very committed management team to invest in such a relatively new endeavor. Since there is limited evidence of results from conducting such experiments, it is difficult to justify the investment in such a pioneering endeavor.

Because of the overwhelming difficulty of reliably measuring process itself, it is even more difficult to attempt to measure the impacts that process change may have on products. Single studies are of very limited value. Pilot projects may be overly biased because of a unique environment (classroom versus production room). Classes of studies may show that a particular process is of no value, and while this may be an interesting result in its own right, it does not produce the result needed by the production organization.

Reviewing work related to the experimental software engineering concept also seems to show that the necessary approach of multiple, repeated, replicated studies is not commonly used. Although there are cases of controlled experiments, and pilot studies looking at some concept of software technology, the willingness to repeat such studies to confirm, or challenge, or further understand a single point is limited. The nature of the experimental software engineering approach is that it requires multiple studies and replicated experiments, but that may not be as attractive to the software engineer as studies that are new and unique.

Each of these points is arguably weak from the point of view of building the knowledge base, but they do pose impediments to production organizations' accepting the value of experimental software engineering. For whatever reasons, the value of these improvement concepts is only very slowly being accepted by the software engineering community.

Lesson 8. Benefits of the EF have been demonstrated early in improvement programs.

It is true that the number of examples of improvement programs applying the concepts of experimentation and product-driven change is quite limited. On the other hand, the examples of successful application of the concepts verify that the value of the program is demonstrated early.

The required elements of the Experience Factory paradigm require that the organization immediately attempt to identify needs and goals as well as to identify current strengths and weaknesses. The organizations providing their experience data for this study indicated that this discipline itself added a tone of improvement to the software discipline.

By requiring the focus on baseline characteristics, models (e.g., cost, defects, activity, test) are available that provide very useful tools to the local organization. This in itself is beneficial near the start of the program.

Common Issues Across Domains

In addition to the eight points summarized above, several common issues were expressed by the organizations providing this experience information. Although the issues were expressed in varying levels of detail and in different forms, all the organizations seemed to agree on the following four major points.

Issue 1. Our ability to characterize 'process' is less mature than anticipated.

Most of the participating groups indicated a weakness in trying to represent the specific processes that were used within a series of projects to provide domain information and to help classify levels of process applied. They noted it was exceptionally difficult to distinguish between projects as to which processes were truly in use and to be able to represent the process elements as distinguishing features. For that reason, there was frustration in attempting to determine which experiences could be shared across domains and which were unique because of specific process characteristics having been applied.

Issue 2. Domain engineering insight has not facilitated sharing of processes.

In the past, a significant effort was made to define and study domains of applications so that efforts to share software code could be accelerated. It was noted that a fair amount of progress has been made in this area and that the ability to share and reuse code has been enhanced through much of this work. The difficulty in establishing the EF or general improvement program points to the need for further domain understanding to identify which experiences, processes, measurement, and lessons can be shared between organizations. The effort to expand the concept of EF has shown the need for a better understanding of domain characteristics, or at least for a means to define these domains for purposes beyond code sharing.

Issue 3. Size or extent of an EF is ill-defined.

As with other improvement programs or rating programs such as the CMM, there is no clear understanding as to the relevance, limitations, and interdependencies of various sizes of particular organizations. Professionals who have attempted to identify larger and larger domains as a single entity for the purpose of having a common process as well as a common improvement program have run into significant difficulties. At present, we do not have insight into the limiting boundaries of size or expanse so that these common processes and improvement programs can be established.

Issue 4. Significant 'uncontrolled' variables impact change and improvement.

Considerations such as people's ability, maturing technology, organizational reengineering, and changing environments have a significant impact on the analysis of improved processes and on our ability to measure them. The validity of observations of process impacts on software products is, therefore, often uncertain. In attempting to measure the impacts that controllable parameters have on the software product, it is difficult to eliminate the consideration of these uncontrolled variables.

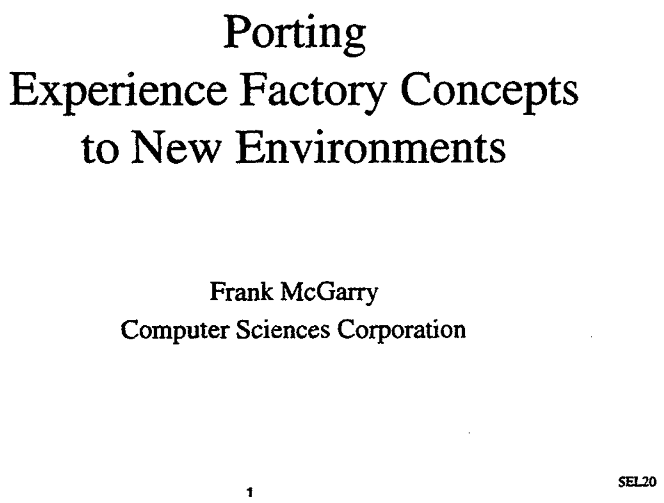
It is a common and persistent question that is posed about why software may be improving. Does the change in our controlled use of process have any impact, or is any improvement completely due to uncontrollable parameters? That question does not appear to have an easy answer, although expanding the types of analysis designed within the EF concept can help address that topic to some degree.

Acknowledgment

The following people provided detailed information relevant to the experiences captured in this paper: Rose Pajerski of NASA/Goddard, Sharon Waligora and Jerry Doland of CSC, Charlene VanMeter of NSA, Michael Deutsch of Hughes Information Technology, Dieter Rombach of the University of Kaiserslautern, and Gianluigi Caldiera and Vic Basili of the University of Maryland.

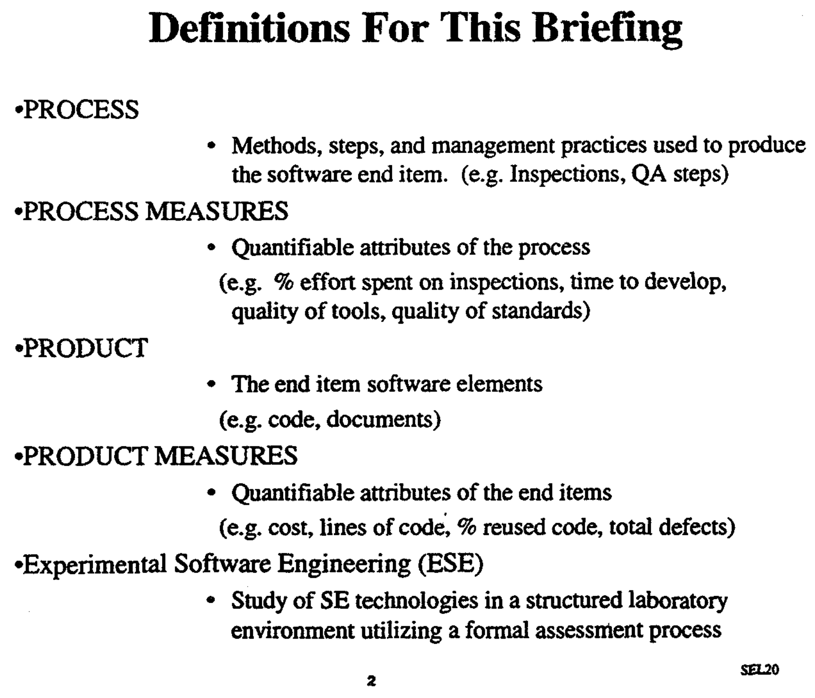
References

1. Basili, V.R., "Software Development: A Paradigm for the Future (Keynote Address)," *Proceedings COMPSAC '89*, Orlando, Florida, September 1989
2. McGarry, F.E., 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. Paulk, M., B. Curtis, M. Chrissis, and C. Weber, *Capability Maturity Model for Software Version 1.1*, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-93-TR-24, February 1993
4. McGarry, F.E. and M. Thomas, "Top-Down vs. Bottom-Up Process Improvement," *IEEE Software*, July 1994
5. Fenton, N., S. Pfleeger, R. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, July 1994



Porting Experience Factory Concepts to New Environments

Frank McGarry
Computer Sciences Corporation



Definitions For This Briefing

- •PROCESS
  - Methods, steps, and management practices used to produce the software end item. (e.g. Inspections, QA steps)
- •PROCESS MEASURES
  - Quantifiable attributes of the process
    (e.g. % effort spent on inspections, time to develop, quality of tools, quality of standards)
- •PRODUCT
  - The end item software elements
    (e.g. code, documents)
- •PRODUCT MEASURES
  - Quantifiable attributes of the end items
    (e.g. cost, lines of code, % reused code, total defects)
- •Experimental Software Engineering (ESE)
  - Study of SE technologies in a structured laboratory environment utilizing a formal assessment process

Characterizing 'Experience Factory'

- **Experimental Software Engineering**
  - Software projects treated as learning instruments
  - Process activities and technologies treated as study variables (not as a priori known solutions)
  - Formal Process Used for change (QIP)
- **Goal driven change**
  - Typically product oriented
  - Product and Process 'baseline' required
- **Separation of Concerns (Specific structure)**
  - EF organization analyzes/synthesizes- produces models/processes
  - Development organization sets goals, characterizes, provides data
- **Measurement**
  - Measurement fundamental at start
  - Primarily used to characterize and guide change

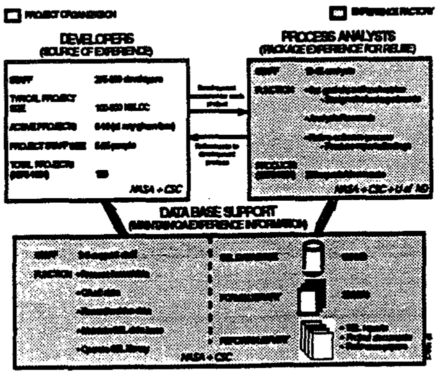
EF Comprises <u>Structure</u> and <u>Activities</u>

SEL20

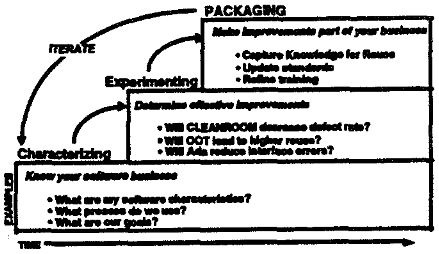
3

The SEL is an Experience Factory

Structure



Activities


PACKAGING
ITERATE
Experimenting
Characterizing
Make improvements part of your business
• Capture Knowledge for Reuse
• Update standards
• Refine training
Determine effective improvements
• Will CLEANROOM decrease defect rate?
• Will OOT lead to higher reuse?
• Will Ada reduce interface errors?
Know your software business
• What are my software characteristics?
• What process do we use?
• What are our goals?
EXAMPLES
TIME

4

SEL20

Basis for the Observations*
(In Addition to the SEL)

- •NASA
  - Center-wide Activities (Goddard, JPL)
  - Agency-wide efforts
  - Several other related efforts (e.g. SEAL at Langley)
- •CSC
  - FDTG(in addition to SEL efforts)
  - SEAS-wide activities
  - Several Centers initiating efforts (ASD in UK)
- •NSA
- •Hughes Information Sciences Division
- •Associates of University of Kaiserslautern
  - (Robert Bosch, Daimlèr Benz, Nokia, Ericsson)

*Only Partial experience data available from each of the above

1 Specific initial products distinguish EF
(4 Major products addressed in first year)

- **•Plan (Document)**
  - Define Organizational Goals
  - Improvement Concepts (learning organization)
  - Approach
  - Target Schedule
- **•Baseline (Document)**
  - Must capture Product as well as Process
  - CMM useful for Process
  - Not Judgmental (but will always be interpreted that way)
- **•Process (Document)**
  - Update, Generate s./w process within organization
  - Establish key practices used within the organization
- **•Program for Measurement (Operation in place)**

1a

Critical elements of s/w baseline

Process

- Key elements of written process
- Major components of Process in use
- Role of management/ support
  - QA/ CM
  - Project structure
- Improvement Activities Ongoing
- Perceptions of developers/managers

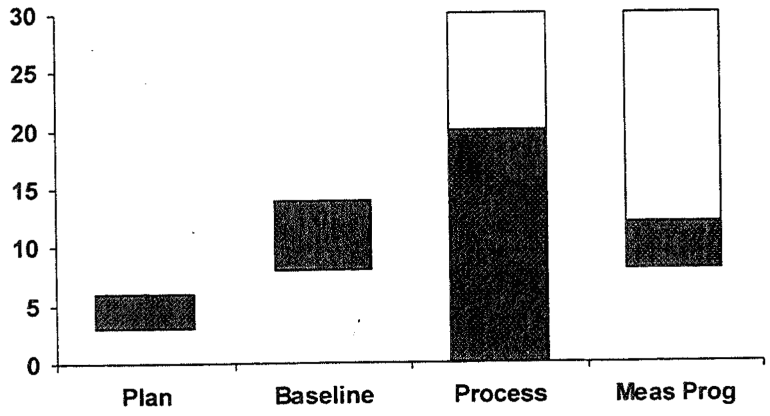
Products

- Size characteristics
- Cycle time
- Cost
  - Total development
  - Maintenance rate
  - By function (QA/CM/Testing/)
- Defects (number and type)
- Defects by test phase/activity

7

SEL20

Start-Up costs insensitive to domain size

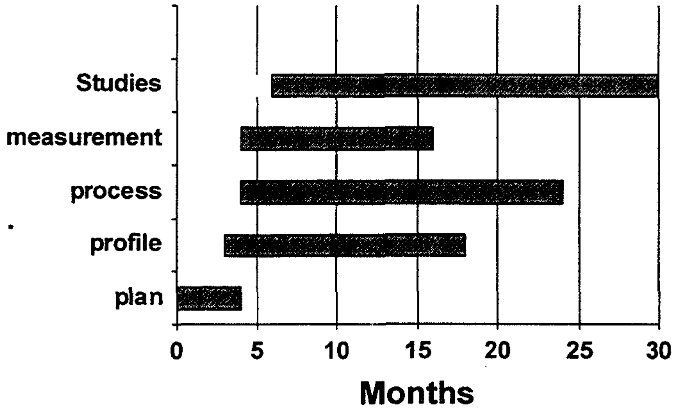


8

SEL20

2a

Timeline for implementation



Startup timeline also similar for all size organizations

9 SEL20

3

Successful operation enhanced by organizational structure

•Separation of concerns exhibited by allocated resources

•Partnership with University - a proven benefit

- Social phenomenon inherent in process ('outside experts')
- Focus of concerns
- Analytical capabilities
- S/W is a laboratory science

•Improvement staff must take on role of 'support' not process developers

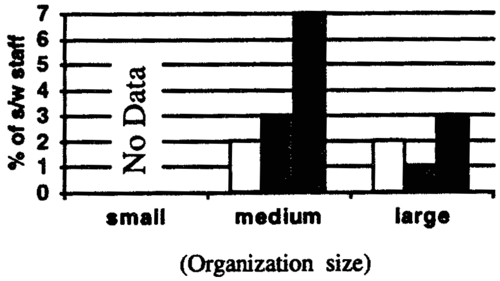
•Rotation of <u>Project</u> personnel through <u>Analysis</u> organization necessary

- Goals and needs must be driven by s/w experience

10 SEL20

4

Operation of improvement program ranges from 4% to 10% of S/W budget



Project Overhead ○
- •Provide data
- •Meetings
- •Training

Information Processing ◍
- •QA data
- •Archive data
- •Basic reporting

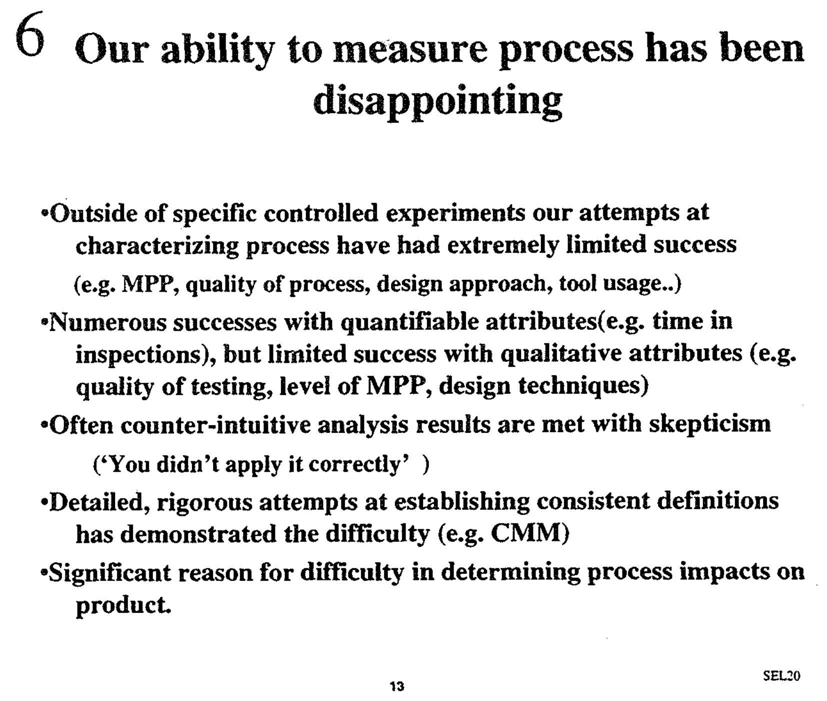
Analysis ●
- •Design studies
- •Analyze/synthesize process
- •Produce new process
- •Provide training

small =	up to 50 s/w
medium=	up to 400
large =	500-3000

5

Process Assessment Models are superb as a tool; They are misleading as a goal

- **•Apply process assessment models to define your process characteristics, not to determine success**
  - **CMM or ISO-9001 are sample tools to support process baselining**
  - **CMM authors (Humphrey, Curtis, Radice) intended them as a guide/not a goal**
- **•Misuse of process assessment models has propagated misdirected efforts**
  - **Can create a false sense of achievement or failure**
  - **Can cause delays in product improvement program**
- **•Operate as Level 5 organization from the start**
  - **EF concept is to improve product by appropriate selection and manipulation of process**
  - **Learn from each project (treat each project as an experiment)**
  - **Plan for process and technology change at start**

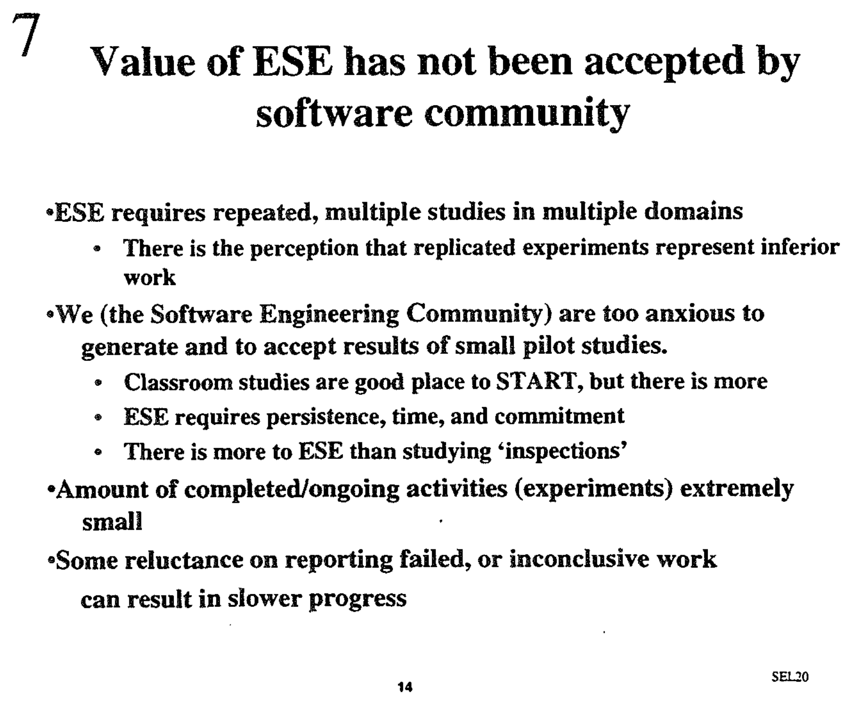


6

Our ability to measure process has been disappointing

- •Outside of specific controlled experiments our attempts at characterizing process have had extremely limited success
  (e.g. MPP, quality of process, design approach, tool usage..)
- •Numerous successes with quantifiable attributes(e.g. time in inspections), but limited success with qualitative attributes (e.g. quality of testing, level of MPP, design techniques)
- •Often counter-intuitive analysis results are met with skepticism
  ('You didn't apply it correctly')
- •Detailed, rigorous attempts at establishing consistent definitions has demonstrated the difficulty (e.g. CMM)
- •Significant reason for difficulty in determining process impacts on product.

13 SEL20



7

Value of ESE has not been accepted by software community

- •ESE requires repeated, multiple studies in multiple domains
  - There is the perception that replicated experiments represent inferior work
- •We (the Software Engineering Community) are too anxious to generate and to accept results of small pilot studies.
  - Classroom studies are good place to START, but there is more
  - ESE requires persistence, time, and commitment
  - There is more to ESE than studying 'inspections'
- •Amount of completed/ongoing activities (experiments) extremely small
- •Some reluctance on reporting failed, or inconclusive work can result in slower progress

14 SEL20

8

Benefits of EF approach can be demonstrated early in improvement program

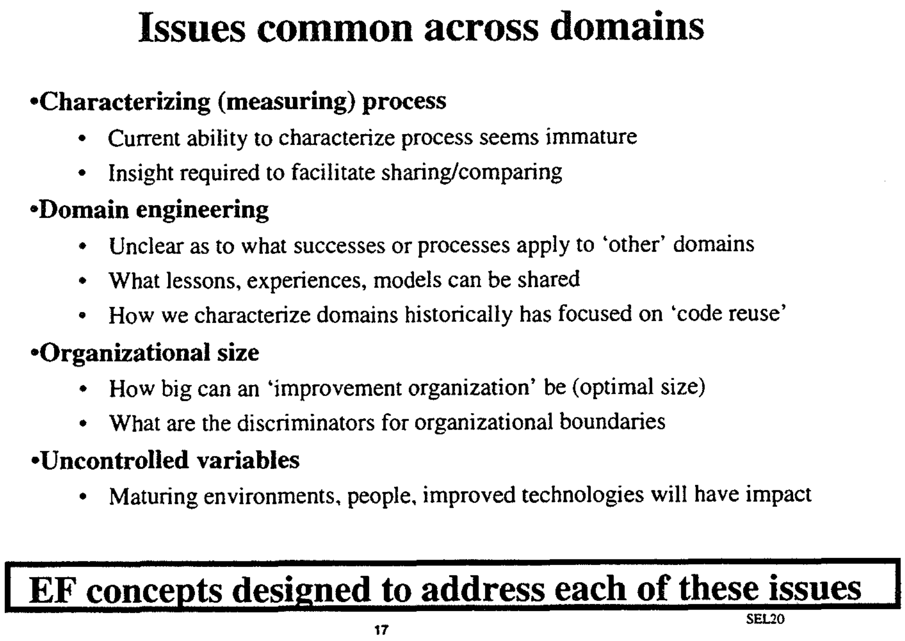
- •Concept inherently enhances capability of demonstrating process impact on product
  - EF focuses on baseline, process, product, and measurement from start
- •Development models, technology management and domain insight natural products of the EF
  - Baseline is first significant step in capturing relevant models (cost, effort, defects,..)
- •Separation of concerns facilities optimization of available resources
- EF concepts helps focus change on solving local domain problems
  - Domain engineering concepts needed to facilitate sharing of experience

Experience Factory Structure and Activities Enable Demonstration of direct and indirect improvements

9

Implementation is very hard

- •Overwhelming inertia still exists toward an easy, one-time solution
  - Pre-defined process attributes seen as complete solution (ISO, CMM, SPICE..)
- •Emphasis on 'Understanding' perceived as unnecessary delay
  - Organizations want action and change- not establishing baseline
- •Concept of 'experimentation' and 'scientific method' foreign to software environments
  - Need for experiments, analysis, evolution deemed inappropriate
- •Changing technology viewed as eliminating need for s/w engineering
  - COTS, Reuse, 4gl, etc. seen as replacements for 'software'



Issues common across domains

- **Characterizing (measuring) process**
  - Current ability to characterize process seems immature
  - Insight required to facilitate sharing/comparing
- **Domain engineering**
  - Unclear as to what successes or processes apply to 'other' domains
  - What lessons, experiences, models can be shared
  - How we characterize domains historically has focused on 'code reuse'
- **Organizational size**
  - How big can an 'improvement organization' be (optimal size)
  - What are the discriminators for organizational boundaries
- **Uncontrolled variables**
  - Maturing environments, people, improved technologies will have impact

EF concepts designed to address each of these issues

17

SEL20

OMIT THIS
PAGE

Session 2: Reliability

Empirical Study of Software Testing and Reliability in an Industrial Setting
Jacob Slonim, IBM Canada

Software-Reliability-Engineered Testing
John Musa, AT&T Bell Laboratories

Reusing Software Reliability Engineering Analysis from Legacy to Emerging Client/Server Systems
James Cusick, AT&T Bell Laboratories

S4-61

416622

360689

An Empirical Study of Software Testing and Reliability in an Industrial Setting

34P

Jacob Slonim
IBM SWS
Toronto Laboratory
North York
Ontario L4W 4P4
Canada

Michael Bauer
Dept. Computer Science
The Univ. of
Western Ontario
London, Ontario
Canada, N6A 5B7

Jillian Ye
IBM SWS
Toronto Laboratory
North York
Ontario M3C 1H7
Canada

Abstract

Today's competitive software environment requires the development of high-quality software in shorter development times. This requirement places increasing demands on software developers, their tools and processes. In particular, since much of software quality assurance still relies on testing, improved testing processes and tools to aid in efficient, cost-effective testing are required. Yet, there is still a great deal that we do not understand of the testing process and the means to assess its effectiveness. In this paper, we report on a study examining the utility of code coverage to support and improve the testing process. The study is conducted during the development of a new release of a large-scale commercial product, but the data collected and analyzed to date are focused on a single line-item (product functionality) of the subjected software. We report on the information collected on the testing of this line-item during the unit and function test phases, what it reveals about testing in this environment, and the tool that was used. We conclude with some lessons learned and directions for additional data collection and analyses to better understand the utility of code coverage in improving software testing in industrial settings.

1 Introduction

In today's competitive climate, a successful software product must be priced competitively and be of high quality (including both functionality and reliability) in order to be successful. Competitive demands in the future will likely require software to be of even higher quality. Current industry practices rely extensively on testing (unit, function/integration and system/acceptance testing)[17] to ensure a software product's or release's quality. Yet testing remains one of the most challenging aspects of software development and often the most costly. The costs associated with testing are very high, relative to the overall cost of product development. Studies report that these costs range from 40% to 50% [1, 20] of the entire product development life-cycle expense (in both capital and time); and, even then are often considered insufficient.

Moreover, competitive pressures often require a software company to quickly release its product to the marketplace in order to protect its investment or market share. Any delay beyond what the competitive pressures of the market might tolerate could jeopardize the product's marketplace acceptance; and in some cases the entire investment could be lost. In the IBM Software Solutions (SWS) Toronto Laboratory, for example, competitive pressures have forced the development cycle to gradually shorten from 18 months to 9 months. This aggressive cycle time requires increased productivity while sustaining quality in all phases of the development life cycle. Such situation is not unique to the Toronto Lab.

The pressure to increase test efficiency is especially high because of the high proportion of total test time during the development cycle[3]. To those charged with function integration and/or product acceptance testing, this pressure increases the challenge of completing their tasks on time as well as ensuring extensive and in-depth testing of the product.

Given the degree to which product reliability can affect the software products and services, it is paramount that our industry conduct testing and reliability estimation in an efficient and cost-effective manner. It is a common observation of many studies[11, 24] and our own experience that the cost of fixing errors grows quickly during the development cycle and more after product release. Hence, it is critical, more than ever in today's environment, to detect errors efficiently and as early as possible in the development cycle. Yet, there seems to be no industry-accepted metrics or test tools of adequate industrial strength to effectively aid testers in determining how their limited time should be allocated in the testing process to improve problem detection and ensure higher software reliability. Based on our experiences, both academic research models and prototypes and commercial tools still fall short of this goal.

Any meaningful estimation using the current software reliability growth models requires, in general, "that the system be well into test before the data required can be collected and the model applied"[5]. In addition, the fact that none of these estimation results provide feedback on how to improve the reliability measurements during the development cycle makes it even harder to justify the often high cost of utilizing these models in a industrial setting[19].

On the tool side, we have not yet found one that has sufficient strength to handle the complex environment and large-scale software often present in industry. Most tools also fail to meet the performance and resource requirements that industry requires. We think the following explanations account for these tools' lack of industrial strength:

- Most tools available have originated from academic studies and, though they may represent interesting "proof-of-concept" prototypes, they are not capable of coping with the level of dynamic use and environment settings, nor the level of complexity in the process model and modules that are often found in industrial software.
- They have been validated and used only on software of much smaller size than many industry products[15]. Many of the tools are designed and used to handle software of tens of thousands of lines of source code; industrial products are often one or two orders of magnitude beyond this (hundreds of thousand or millions of lines of code).
- They are neither reliable nor suitable for use on a daily basis; they break down frequently, and/or do not handle abnormal conditions well. They do not cope with a range of source

code formats and language variations, nor operate over a range of development phases, from the unit-test to function test, integration, and system-acceptance test.

- They often require significant resources and not perform efficiently enough to be accepted as part of standard practice during development phases.

As a result, there are no software reliability growth models or tools in widespread use within the Toronto Laboratory. Some work had been done[21, 22] in trying to use execution-time based software reliability growth models at the very end of product development cycle—the system test phase; however, in general, especially during the early phases of the development cycle, software product reliability is "estimated" by the number of defects found during system tests and, very importantly, the "sense" of developers and testers towards the product. This is not necessarily bad, but it just highlights the fact that there are inadequate tools to provide quantitative evidence to support their feelings, nor are there systematic methods to provide evidence that sufficient tests have been done[4, 23].

It is our observation that developers and testers are genuinely interested in producing highly reliable software products, even though they are, in general, also striving to meet deadlines. They constantly search for feedback in perfecting their work and are willing to use any tool that would help them do high-quality work more efficiently. Software professionals have suggested that the following information, if available early in the development cycle, could provide them with useful feedback[16]:

- the quality of the manually and automatically generated unit, function and integration test cases;
- the code covered by tests performed, especially on the newly added or changed parts of the code;
- the quality of the regression test cases and system-acceptance test cases, especially of the most critical components or functionalities.

It is not feasible to collect this information manually. Tools to collect this information must be integrated into the development environment and work seamlessly; otherwise, they will probably be ignored because of the time constraints.

Many assumptions (implicit and explicit) in the development process and current practices involving testing and software reliability measurements and criteria have not been validated. The subject of software reliability growth estimation and reliability assurance while the software is under development, especially in an industrial setting, has not been very well studied. From our initial observations, code coverage is a promising measurement technique for assessing the quality of testing, which may lead to more reliable estimation, particularly in cases of the absence of an accurate operational profile. Compared to measurements that are commonly used in other software reliability estimation approaches (such as number of test cases run and time of testing), source code coverage approach has the potential to add a new dimension to the development life-cycle by providing an "inside out" view of the tests, otherwise unknown to developers, in a timely manner. We hope such feedback provided in the early stages of the development cycle can be used to improve product quality efficiently, reduce the time and cost required to get the product into the market, and provide valuable information for software project management. It is the objective of our study to begin to systematically explore these issues in an industrial setting, hoping this research will lead to a new generation of software reliability assurance models, measurements, tools, and processes.

Most of the serious industrial studies in the software reliability area to date, are conducted by telecommunication organizations[12]. It is our intention to follow their lead and take advantage of their approaches, findings, and experiences to further explore practical software reliability issues in industries outside telecommunications.

This study is part of an ongoing project of the IBM SWS Toronto Centre for Advanced Studies[18] in conjunction with Bellcore, The State University of North Carolana, Purdue University, and the University of Texas. As a first step, we wanted to gain more accurate measurements about the tests performed during product testing by using a coverage monitoring tool called ATAC (Automatic Test Analysis for C), developed by Bellcore. The tool records which lines of source code were executed when tests are run (whether through manual or automatic execution of test cases/scenarios) and reports the total code coverage accumulated. We collected defect information along with the actual testing and defect removal practices during the development of a major new release of a large-scale commercial product. The development and test team followed the Lab's usual software development process and were unaware of this study. This data will be analyzed to measure current development and testing practices and to gain a first-hand understanding of them. We will try to answer questions such as : What is realistic code coverage to expect from a typical industry software development practice in various phases? Subsequent work will involve more experiments to identify what improvements could be made if such tools were available during the development phase, and how much that would affect results. As the first phase of this study, the actual coding-and-unit-testing and function-verification-and-integration-testing data for one product functionality in this new release have been collected and analyzed.

The rest of the paper is organized as follows. Section 2 outlines the development environment in which this study is carried out. Section 3 describes the coverage tool used. The experiment and analysis of data collected are presented in Section 4. Some things we learned about testing and the development processes during our experiment to date are described in Section 5. The paper concludes with a summary of the work and some future directions.

2 Environment

The Toronto Laboratory's development process is based on the the spiral model and waterfall model of software development[9, 13]. An external function specification is first constructed based on customer requirement and product planning. The development team develops a design based on the external function specification, writes high/low level design documents and has them reviewed and approved (revised and re-reviewed, if necessary). Then this design is implemented and unit tested. After unit testing, code review (again, making changes and re-reviewing, if necessary), and some basic common (product-wide) integration tests, the function is integrated into the base system. Meanwhile, a function-specific integration test plan and test cases are written and reviewed. Once the function is integrated into the code base, the integration test phase starts; test cases are executed and bugs fixed. At the integration test exit point, the test cases are added to the regression bucket and constantly rerun on the product. Once all the functions for a release have passed their integration test, the system-acceptance test phase starts. Test cases are run according to the system-acceptance test plan until the exit-criterias are met.

Note the above is only a sequential description of what is actually an ongoing, cyclical process. Even though it is a well-documented and well-practiced procedure, there are little systematic or scientific measurements on the quantity and quality of the tests performed, because of the lack of feasible tools. The assurance of the test quality is mostly based on the review of the test plan and test cases, and the tester's "gut feel" about the sufficiency of tests performed to date.

As noted, our experiment took place during the development of a major new release of a large-scale commercial software product. We chose to conduct our study on a major new release for two reasons. First, the modifications and changes are more significant than those of a (minor) refresh of the software; hence, a major release provides extensive enough code changes to make an interesting case for our study. Second, it is also much more representative of industrial software development, in our opinion, than the creation of a brand new product, since most of industry software practice involves reusing old code[2].

The release of the software under consideration involved adding new functionality and moving into parallel systems and operations. There was a common code base with a variety of versions and platforms that had to be maintained. This obviously complicated the changes and testing. The base software (that is, the old product) contained approximately 420 KLOC (thousands of lines of code), and this new release added or changed approximately 150 KLOC (all code counts in this paper do not include comments; one source program statement is counted as one line). Code in this product is subdivided into about 40 components according to their functionalities and interrelationships. Product developers are divided into small development teams (usually two to five people), each responsible for a certain line-item. (**"Line-item"** is a term for a set of related tasks that evolved in the development of, typically, a product functionality). Each development team's tasks include designing, coding, unit-testing, and conducting reviews and function-integration testing of the line-item. At the end of product development, all line-items are integrated together for intensive system-level acceptance testing conducted by a separate test group. The implementation of a line-item typically involves changing multiple old source modules (each often containing multiple functions) and adding new modules in multiple components.

A collection of all the source code needed by the product is often called the **"code base"**. When the project is started, the old product is the code base; the base changes and grows as development progresses. A collection of all the existing (automated) function-verification-and-integration test cases of the product is referred to as the **"regression bucket"**; it grows when new functionalities are added to the product.

Because the regression bucket of this product was too big to be completely executed within a day, a small set of test cases were carefully developed and selected from the regression bucket to be used as a basic product-wide common test suite, referred to as the "basic integration test" or **"fastpath"**. Whenever the code base was changed (usually every day), it was compiled, linked, and verified by the "fastpath" test, then regression tested.

A verified runnable base is often referred to as a **"build"**. A common build is the base that is accessible by everyone in the project. A private build is a snapshot of the common build plus one's own modifications not yet made public to others. Code changes (for defect-fixing or for line-item implementation) are required to be made on a private build first, then unit tested. They must pass the "fastpath" before being added (checked-in) to the common code base. This "fastpath" testing and checking-in activity is also referred to as the **integration** of a defect fix or a line-item. Once integrated and verified, the change becomes part of the new build.

In this environment, source files in the base are shared and updated by all developers as needed; there are always multiple line-items and/or defect-fixing activities that take place in parallel and often several developers need to make modifications on the same source modules. Exclusively locking a source file for too long (more than a day) often means delay in other people's work. Developers working on related components or line-items are often involved in joint discussions or are asked to review each other's work in order to avoid design or implementation conflicts and to maintain the global coding consistency of the product.

Work on a line-item often lasts more than several weeks, which means it is quite impossible to reserve exclusive use on all the needed files during the whole period; hence, sources in the private build may be inconsistent with what is in the common build. Therefore during the coding-and-unit-testing phase of a line-item, the private build is periodically updated with the latest common build to make sure that the implementation in progress keeps up with the latest changes in the code base; this activity is also referred to as a **merge**. A line-item typically involves more than one merge from the start of the implementation work to its integration. Coding, review, unit testing, and changes are ongoing activities on all these periodically merged *private* builds.

The line-item's source files are exclusively locked only during the integration of the code. The integration of a line-item locks all the needed files, makes the final merge, reruns all unit tests, passes "fastpath" test (and other verification tests when applicable), then checks-in the code and releases the locks. In general, all review, unit-test, and "fastpath" defects have to be fixed before the code integration, which marks the end of the coding-and-unit-testing (often simply called "unit test" or **"UT"**) phase of the line-item.

From then on, the function-verification-and-integration-testing (often simply called "function (verification) test" or **"FVT"**) phase of the line-item starts. The line-item's specific test cases, as well as test cases developed for all other line-items and the regression test buckets, are run daily on the *public* builds. Once all these function-specific test cases pass without errors, the function-verification-and-integration-test of the line-item exit; the test cases are added to the regression bucket. Then the system-verification-and-acceptance-testing (often simply called "system (verification) test" or **"SVT"**) phase starts; system test cases, as well as all the regression test cases are executed daily until the product **GA** (general availability).

For this study, we chose to focus on a line-item that was developed by a five-person group. About 5 KLOC were changed or added to approximately 50 modules (source files) in five components. These 50 modules entail about 35 KLOC and the five components contain about 150 KLOC in total. The team followed the standard in-house process for development. They were, in general, unaware of this study although they were asked to keep records for all defects found in all development phases. This procedure was necessary because defects found during the coding-and-unit-testing phase are not usually tracked, but were important for our study of the early development cycle.

To give a sense of the code complexity of the subject software, Figure 1 summarizes the cyclomatic complexity of the more than 250 functions that existed in the modules involved in the subject line-item. The cyclomatic complexity is McCabe's metric[10] on control flow complexity. Note that it only reflects the complexity of individual functions, not inter-modular or inter-functional complexity.

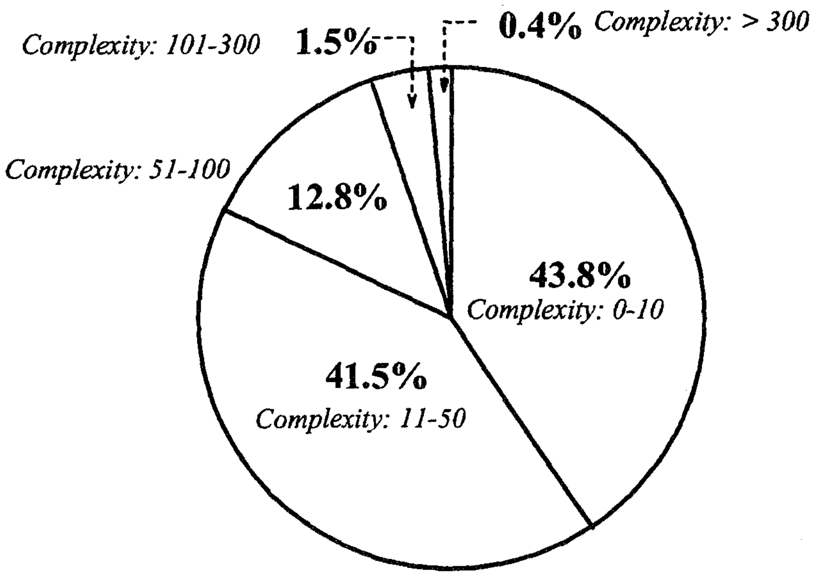


Figure 1: *Percentage of Functions with Various McCabe Complexities*

The average cyclomatic complexity per function in this particular line-item was 23.18. In general, a function with a cyclomatic complexity of greater than 10 is often considered "too complex". There were a few functions having a complexity over 100. In most instances these functions had evolved over several generations of the product, and the original developer(s) have moved on to other projects. There is a reluctance to spend the time or risk introducing new coding errors to rewrite these modules because of their complexity and the fact that their defect rate is not higher than normal.

3 The Coverage Tool

The source code test coverage tool used in the experiment is called ATAC, a tool originally developed at Bellcore, then augmented at Purdue University to be used in our environment.

ATAC stands for "Automatic Test Analysis for C"[7, 8]. It measures how thoroughly a program is tested by a set of tests using data flow coverage techniques. Structural coverage testing identifies program constructs (*attributes*) that may be exercised during program execution and determines which of these constructs are in fact exercised by a set of tests. These constructs may be *blocks* of consecutive statements, branch *decisions*, or various combinations of assignments and uses of variables (e.g., the *all-uses*). The report generation function of ATAC provides a summary of the percentage of testable attributes being executed, or shows the details of exactly which lines of the code are exercised by a set of tests. It can be used to identify overlap among test cases, and areas of source code that are not well tested.

Although the original version of ATAC had to be modified in order to handle the complex environment — one involving multiple processes, parallel systems, concurrent users, and various applications, it is by far the most reliable and usable tool that we have tried. Its architecture also makes it the most suitable for easy customization to our environment.

At this point, ATAC still has limits on scalability, functionality, and performance that restrict the scope of our experiment. However, we do see its potential in becoming a reliable industrial-strength software testing and reliability assessment tool. IBM SWS Toronto Centre for Advanced Studies is conducting a joint project with Bellcore in achieving this goal.

4 Experiment

This experiment involved revisiting the line-item, the test cases, the recorded defects, and their detection and removal history. Subsequent to the release of the product, an experiment of the utility of the ATAC tool had begun. The objective of the experiment, which continues, is to understand the potential role that such a test code coverage tool can play in the early detection of defects, and how it may provide quantitative information on the testing process to developers and testers.

In this section we describe a case study on a single line-item's unit-test and function-test data. Subsection 1 briefly describes the development cycle of this line-item. Subsection 2 outlines what type of data was collected, how they were collected and analyzed. Subsection 3 and 4 show the data collected from the unit-testing and function-testing, respectively. Then in subsection 5, we report our observations on the combine the data from the two testing phases and make some analysis.

4.1 The development cycle of the line-item

The overall development cycle of this line-item (Figure 2) lasted approximately 13 months. Of these, the design took about 3 months, and the coding-and-unit-testing took about 3.5 months. The function/integration and acceptance testing took the remaining 6 or more months. The formal entry of the system test is also the time for product code freeze; after that date, only defect fixes can be integrated into the code base, not additional line-items. In this particular case, this test lasted approximately 2.5 months.

Note although included as part of the line-item's development cycle, system/acceptance test is a product-wide activity, which does not focus on any particular line-item as unit-test or function-

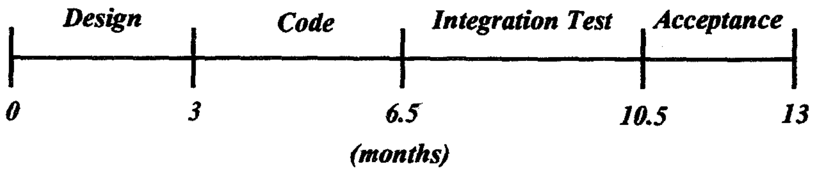


Figure 2: *Development Cycle*

test normally does.

4.2 How the data are collected and organized

We collected all the defects on the subject line-item that were detected and fixed before the system-test started. These defects were then analyzed according to their error-characteristic and detection information.

A "defect" in this paper is a logical concept (not interchangeable with software "fault"). That is, a defect may involve a fix to one statement or to many blocks in many source modules, as long as these changes are logically closely related to one idea or problem. For example, a design error that counted as one defect may involve changes in many parameters, functions, and files. An initialization defect may appear in one or multiple functions and files, and, as long as they are all the same, is counted as one defect. A failure may be caused by multiple defects, and the same defect may cause various different failure symptoms.

We consider "severity", "risk", and "error-type" all part of a defect's error-characteristics.

- Seven error-types were used to categorize the defects: (1) *logic/control errors*, (2) *interface errors*, (3) *data errors*, (4) *computation errors*, (5) *initialization errors*, and (6) *design errors/problems*, or (7) *code omission errors.* The first 5 categories are based on a study by NASA[1]; we added the last two types because the coding-and-unit-testing phase stage was not part of that NASA study.
- In addition we have classified the defects by *severity* and by *risk* based on our experience and insight into the particular product. Risk is an assessment of how likely a user would be to encounter the particular defect; it can be *low, medium*, or *high.* Severity is estimated on a scale from 1 (very severe, meaning the program dies or does not function as specified) to 3 (low severity, for instance an imperfect message).
- We will use summary tables to report error-characteristics of these defects in the next few subsections. If multiple levels of severity or risk were presented in the defects of the same type, they are listed according to their frequency; that is, the severity or risk level with the most number of defects is listed first.

Defect detection information includes details on how was the defect reported and the cost of locating and fixing it.

- In this experiment, defects were reported from: (1) *formally conducted code reviews*, (2) *casual reviews, while implementing the line-item, or investigating and fixing other defects,*

(3) *generic testing*, such as "fastpath", (4) *regression test cases*, (5) *test cases specifically generated for this line-item*, and (6) *test cases generated for other line-items* (those line-items were developed in parallel with, and typically related to, the one under our study; their test cases were not in the regression bucket because they had not completed their function test at the time).

Most of this information was available from the defect database (for FVT), some was obtained from the line-item's developer (for UT).

- The cost of finding a defect includes time to set up the test environment, run the test, collect debugging information, recreate the problem, etc. work continues until the problem is located in the code, or clearly identified. The cost of fixing a defect includes time to design, code and review the fix, unit-test it, and verify it with the original failure situation(s).

  This information came from a combination of defect records, reconstructed test situations and estimations from developers directly involved in fixing the defect.

  Hardware or software required to run the test were not counted in the cost. For most cases (especially in the FVT phase), time to generate test cases was also not counted as part of the cost to find a defect, because a set of test cases were usually designed and created together. Also, the error detection rates of all test cases in the set typically vary greatly; hence it is hard to get a fair estimation of test-case-creation-costs per defect or per test case.

4.3 Report from the Unit-Testing

4.3.1 Source of defect-detection

As described earlier (Section 2), the coding-and-unit-testing of a line-item is performed on private builds. The implementation starts with a snapshot of the common build as the base; code is constantly added, internally reviewed, tested and changed. The internal reviews were done by the team members who work directly on the line-item; they were casual ongoing activities throughout the entire coding-and-unit-testing phase. When the coding was completed, external reviews were formally conducted at code-review meetings. They involved developers both from the team and others who had knowledge of related components but were not members of the team for the subject line-item.

In this particular case, out of all defects found in this phase, 25% of those defects were found through internal reviews, with an average cost of 3.1 hours per defect; 42% were found by external reviews, averaged at 3.9 hours per defect; and a total of 33% defects were identified through the use of test cases, with an average cost of 5.9 hours to find and fix.

This study shows, in agreement with the findings of others[6, 1, 14], that code review is a highly effective approach for the quality assurance of software. More importantly, its value derives not only from the fact that it finds a higher number of defects with smaller effort than usual test methods do, but even more from its discovery of many defects that would be (from our analysis) very difficult to detect using test cases.

4.3.2 Testing and source code coverage growth

We realized that it was a mistake not to collect information about test cases that run successfully during the unit test phase. Since only test cases that experienced failures were recorded; hence, the coverage measurement shown here does not reflect an exact picture of the actual testing (that is, compare to the actual situation, the total coverage shown is smaller and the cumulative-defect-verses-coverage shown is sharper). However, we think the defect and coverage growth pattern provided by ATAC based on the currently available information still yielded some valid and refreshing results.

Figure 3 shows the cumulative percentage of defects detected and removed plotted against the cumulative percentage of source code coverage by the unit-testing for three different coverage attributes: *blocks*, *decisions* and *all-uses*.

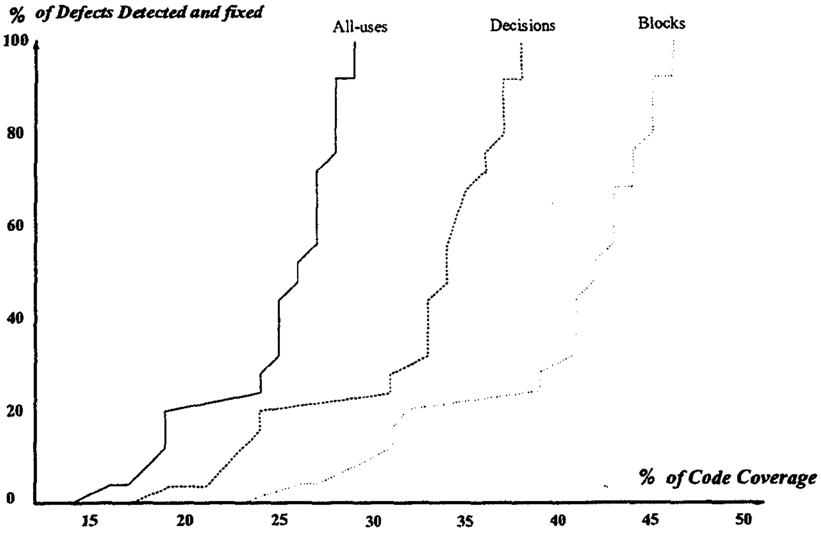


Figure 3: *Code Coverage in the Unit Testing Phase: Three Attributes*

Note that, first, the *Percentage of Defects Detected and Fixed* in Figure 3 refers to defects actually detected (by test cases) and removed during the UT phase of the subject line-item. Hence 100 percent defect removal here does not mean there were no more defects found later during the FVT phase of this line-item. Second, these defects were recorded during the actual UT phase by developers following the standard in-house procedure without the aid of the ATAC tool; all the coverage measurements were later obtained based on the recorded test/defect information on a code base that had all these defects removed.

With these facts in mind, there are several interesting observations that can be made about the data:

- The same test cases provided a much smaller percentage of coverage when presented by the *all-use* attribute than by the *decisions* attribute, because the different coverage attributes represent different granularities of measurement and, therefore, result in different numbers of potential variations. To be exact, there were about 19,000 blocks, 13,000 decisions and 53,000 all-uses variations in the 50 or so modules we studied. This demonstrates the meaninglessness of describing the code coverage achieved by a set of test cases without identifying the coverage attributes used.

- When the first defect was detected, the test cases run up to that point had accumulated a coverage of 50% by function (131 out of total 260), 26% by blocks (4944 out of 18897), 19% by decisions (2445 out of 12601) and 16% by the all-use attribute (8389 out of 52906).

  This high first-defect coverage value, and the flat-slow-sharp defect-versus-coverage growth pattern (common to curves of all three attributes) can be explained by the effect of code reuse. It suggests that code reuse is a reliability-wise method for software development, as seen by many researchers[2].

  On the other hand, it also suggests insufficient testing (including long hours of test case execution that do not increase code coverage) may lead to incorrect conclusions on software reliability, since there may be no, or few defects found until some threshold of code coverage is reached; although we believe that absolute value of this threshold may vary, among different line-items, releases or products.

- The curve for each of the attributes is rising sharply at the end of the execution of the available test cases. This indicates, we think, a need for more testing.

  Although we have explained that the actual curve would not be as sharp as shown due to the successful test cases that were misrecorded, we believe the growth pattern should remain similar for this unit-test phase, which means additional test cases would likely be beneficial.

  This finding is a very encouraging one. Normal measurements (for example, the defect detection rate versus the number of test cases run or the test execution time) had given an opposite impression to this at the end of the unit-test phase (based on the fact that, all recorded test cases, "fastpath" and a lot more generic integration tests were rerun after the last merge before the integration of this line-item, no failures occurred). This result suggests that the use of a code coverage tool may provide a new perspective to aid in pushing more efficient tests to an earlier point in the development cycle, where it should be much more cost-effective to remove software defects.

4.4 Report from the Function Testing

4.4.1 Source of defect-detection

Once a decision is made that the coding and unit-testing is completed, the development process moves into the function-verification-and-integration-testing phase (see figure 2). The objective of this phase is to bring together the newly developed and changed code for this line-item with changed made by other line-items and the existing unchanged code base. Although tests are still line-item oriented, the test environment is shifted to the public build, where regression tests and all pre-FVT-exit line-items are run in parallel, and the code base is updated constantly with new functionalities and defect-fixing changes.

In this particular case, out of all the defects detected (and fixed) for this line-item during the function-verification-and-integration-testing period, 30% were found by test cases specifically generated for this line-item, with an average cost per defect of 23.2 hours (not including time spent planning and generating the test cases); 30% were found as side-effects while running test cases generated for other related line-items, at an average cost per defect of 20.4 hours; 12% were detected while running regression buckets and other generic tests with an average cost per defect of 14 hours; and, most striking of all, the remaining 28% of the defects were found while working on (that is, code-inspections introduced by investigating and fixing) other defects at an average cost of 3.9 hours per defect!

Three points can be learned from this result:

1. Even in the function test phase, it is still extremely beneficial and cost-effective to do as many code reviews as possible.
2. The sources of defect detection and the cost distribution suggest that it is more cost-effective to reuse existing applicable test cases than to create new ones. For instance, test cases created for other line-items and those in the regression bucket can be considered existing ones, from this line-item's point of view. ATAC can be used to limit the creation of new test cases that do not provide additional coverage beyond those test cases already in existence.
3. Except for the defects found by code inspections (whose costs stay about the same level as during the unit-test phase), the cost to fix defects found by all other means are much more expensive in FVT phase than in the UT phase.

4.4.2 Testing and source code coverage growth

Ideally, since function tests are carried out in an integrated environment, and all line-items and regressions are run on a shared public build, defects on all related line-items (whose work overlap of the set of source files each modified) should be collected and analyzed together, then plotted against the total coverage collectively achieved by all test cases from these related line-items and applicable regression test cases, to completely reflect the actual situation. However, limited resources have not allowed this approach as an option for our study to date.

As the first step of the study, we decided to continue to focus only on the single line-item we did the UT experiment on, and follow it through its development cycle. That is, we would analyze defects on this line-item and would only collect coverage information for the complete set of test cases that were specifically generated for this line-item. Test cases generated for the testing of other line-items were each only included in our data if it revealed a defect on the line-item under study.

We understood such strategy would cause the defect-versus-coverage curve to be artificially sharp, but hope this limited experiment would still give us enough insight on how the utility of code coverage can assist in the assessment of the function test process, and help build a interesting case for continuing this study in bigger scope in the future.

Figure 4 illustrates the cumulative percentage of defects detected and removed plotted against the cumulative percentage of source code coverage by the FVT test cases for three coverage attributes: *blocks*, *decisions* and *all-uses*.

Similar to the unit-test figure, keep in mind that the *Percentage of Defects Detected and Fixed* in Figure 4 refers to defects actually detected (by test cases) and removed for this line-item

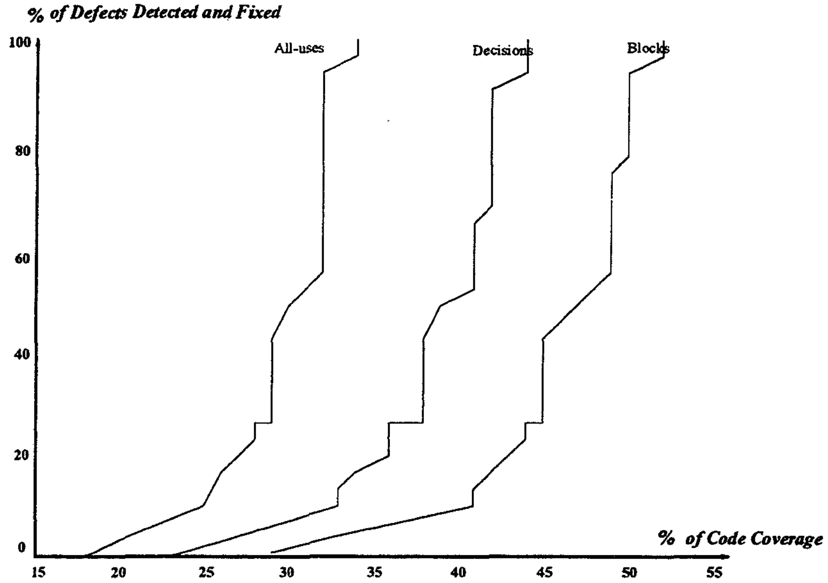


Figure 4: *Code Coverage in the Function Testing Phase: Three Attributes*

during the function-verification-and-integration test phase. All the test cases were rerun after the product release on a build with all these defects fixed. 100 percent defect-removal in the figure does not mean there were no more defects found in a later test (SVT) phase of the product. We then have the following observations:

- In contrast with similar data for unit testing (see Figure 3), note that function testing has a higher "first defect coverage" percentage value. This is because some defects have already been found during unit testing.
- Again contrast this with the unit test data (Figure 3), note the FVT curves for all coverage attributes start their sharp growth at the value where the unit test curves stopped. For instance, by all-use, the unit test was terminated at 29%, which is exactly where the function test curve starts shooting up. the same is true with the other two attributes. This confirms our thoughts when we finished analyzing the unit-test data, that the unit-test curve of this line-item suggested that additional tests would be beneficial.
- It is interesting to notice the different impression we get from looking at the detailed defect-and-test-case report against the curves generated from the same report. In the detailed report, we see many test cases (specifically created for this line-item) increased code coverage without encountering defects. From this, one might expect to see some flat lines on the figures; however, there are hardly any horizontal lines at all. On the contrary, most lines are vertical, as if defects can be found at that point with almost no effort at all.

  This reveals a fact that along the testing progress, it becomes harder and harder to increase a percentage in code coverage, because the part of code not yet covered are left in deeper and deeper branches. Each additional test case is typically able to reach fewer and fewer additional branches of code, and therefore more and more unique test cases are needed in order to increase a certain coverage attribute by 1 percent.

This does not mean that the growth of defect is no longer correlated with the growth of code coverage. In the detailed report where the actual counts of coverage attributes (not just the percentage values) are presented, we still see that all tests that detected new defects increased code coverage.

4.5 Observation and Analysis on the Combined UT and FVT Data

In this section, we first present the combined defect and coverage information from the code-and-unit-testing and the function-verification-and-integration-testing phases of the subject line-item. Comparisons on the error characteristics for defects found and fixed in these two phases are made, based on type, risk, severity and cost. We then analyze these data, the data collection methodology, and the underlying development and test process for answers to the following questions: (1) Why do we seem to hit a wall around the 50% point in block coverage? (2) Why were defects found in FVT after UT was done? (3) Why were FVT defects cost more to find and fix?

4.5.1 The combined UT and FVT defect-versus-coverage data

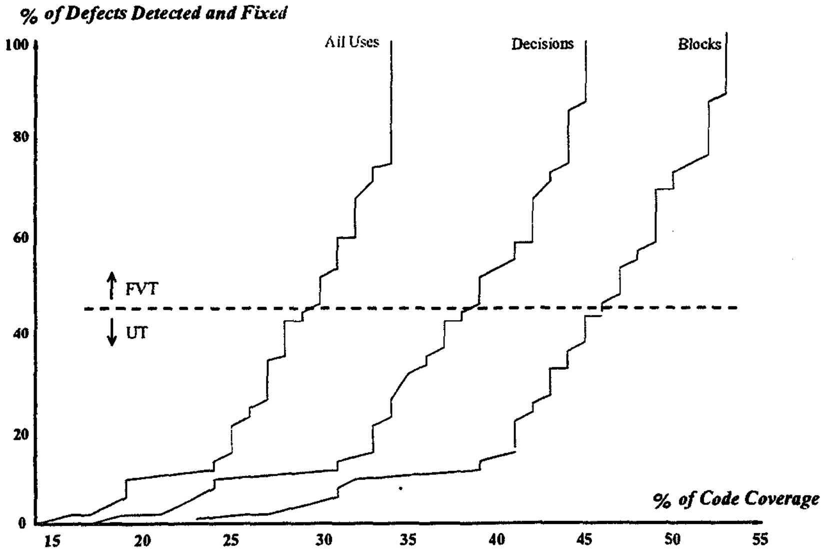


Figure 5: *Code Coverage: Combined Unit and Function Testing Phases*

Figure 5 illustrates the cumulative defects found in UT and FVT phase versus and cumulative percentage code coverage on the 50 or so source files changed or added by the line-item, when test cases from both unit testing and function testing were combined.

Despite the fact that our data collection strategy (as described in the previous subsection) may have artificially sharpened the growth curve in some degree, We see from Figure 5 that from UT to FVT of the subject line-item, the additional percentage of code covered is relatively small, still, about the same number of new defects were detected through the FVT test cases as the UT test cases.

This observation suggests that there was substantial overlap among the UT and FVT test cases from this line-item's code coverage point of view. Further, the amount of new defects detected by these FVT test cases suggest two more possibilities: first, the same test cases run in a changed or more complicated environment (integrated verses private) may reveal additional defects; second, a more complicated test case that covers a combination of several previously tested simpler test cases may reveal new defects.

4.5.2 The UT and FVT defect characteristic comparison

Table 1 summarizes all defects (including defects found by review or fixing code) on this line-item found during the coding-and-unit-testing phase and the function-verification-and-integration-testing phase according to their error characteristics (that is, error type, risk, severity, time to find and fix, and how they were detected).

In both testing phases, about 30% of the defects were found by test cases specifically designed for the line-item, and about 30% were found while coding the line-item or investigating and fixing other problems. The remaining 40% of the defects in unit testing phase were found through formally conducted reviews, and the remaining 40% of the defects in function testing phase were found through regression testing or from test cases created for other line-items.

Figures 6, 7, and 8 graphically compare defects' distribution and cost (in terms of time) between the two phases, categorized by error types. One can readily see from these figures and tables that defects detected and fixed in the FVT phase are much more costly (approximately 3 to 4 times so, on average) than those found and corrected in the UT phase. This observation is not new; what interested us was what could be learned from these data to answer the following question: What could be done to reduce this cost, and at the same time ensure software quality and accelerate the development process?

Table 2 looks at the problem from a different perspective. While summarizing the defects detected in the two phases according to their severity and risk, it reveals that those found during the UT were, in general, more critical (higher severity and risk) than those detected during the FVT phase. This suggests that defects from these two phases should not be treated as equal; that is, while it is wise to have a clean cut on unit-test defects before integrating the code, it may be a smart move not to wait until all function-test defects (non-critical ones) are cleaned up before starting the system test, so that the overall process can be sped up.

4.5.3 Why do we seem to hit a wall around the 50% mark in block coverage

"Walls" seem to exist at a low coverage value (50% for block, 45% for decision and 35% for all-use) on all the defect-coverage curves in our experiment. These walls confused us until we realized the problem was caused by the mismatching between the focus of the set of test cases and the set of source files we chose to measure the defects and coverage upon. A collection of all source files that changed or added by this line-item was the base for the study. Within these files, there existed many functions that were not changed or even used by the new product-functionality implemented by this line-item. The new functionality was the interest of the unit-test and function test of this line-item, therefore the UT and FVT test cases generated for this line-item

	Defects Detected in the UT phase					
Defect Type	Count%	Severity	Risk	Avg. time	Avg. time	Detected
				to Find(hrs.)	to Fix(hrs.)	by
Computation	6.7	1, 2	H, L, M	1.6	2.0	RW, ST,CF
Data	15.6	1, 2, 3	H, M, L	2.8	1.8	ST, RW, CF
Design	1.1	2	H	4.0	24.0	RW
Initial.	13.3	2, 1, 3	M, H, L	1.5	0.9	RW, ST, CF
Interface	14.5	1, 2, 3	H, M, L	1.7	1.7	CF, ST, RW
Logic/Contr.	23.3	1, 2, 3	H, M, L	2.8	1.3	ST, RW, CF
Code Omission	25.5	1, 2	H, M, L	2.4	2.8	RW, CF, ST

	Defects Detected in the FVT phase					
Defect Type	Count%	Severity	Risk	Avg. time	Avg. time	Detected
				to Find(hrs.)	to Fix(hrs.)	by
Computation	4.6	1, 3	M	11.0	7.0	RT, CF
Data	14.0	3, 1	L, M, H	10.5	5.0	OT, CF, ST
Design	4.6	1, 2	M	7.0	15.0	ST
Initial.	23.2	3, 2, 1	L, M	6.3	5.0	ST, OT, CF
Interface	4.6	3	L	1.0	1.5	CF
Logic/Contr.	35.0	3, 2, 1	L, M, H	7.8	4.4	CF,OT,RT,ST,GT
Code Omission	14.0	1, 2	L, M	21.0	14.0	OT, ST

Legend: **How Defects Were Detected**	
RW:	Review (formal code-inspection meetings)
CF:	Coding and defect-investigation or fixing activities
ST:	Test cases specifically generated for this line-item
OT:	Test cases generated for other (related) line-items
RT:	Regression tests
GT:	General (integration) tests

Table 1: *Comparison of Errors Detected during UT and FVT*

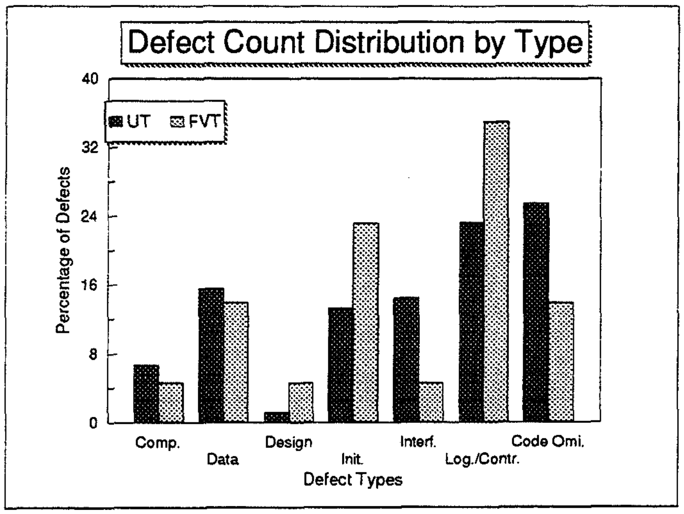


Figure 6: *Distribution of Defect Counts by Type in the UT and FVT Phases*

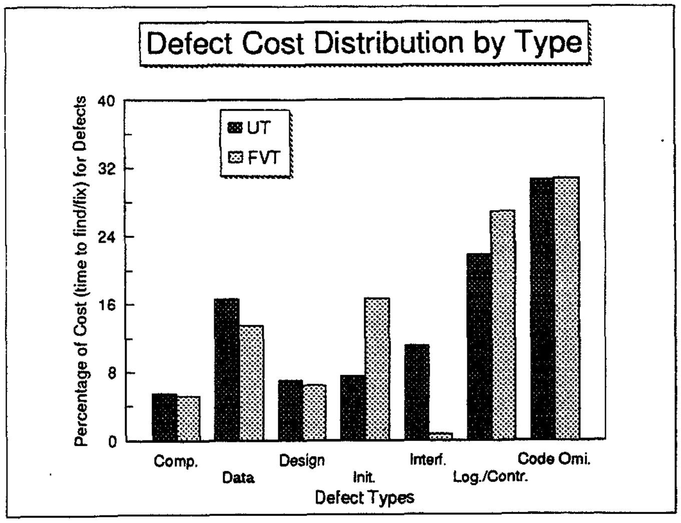


Figure 7: *Distribution of Defect Cost (time) by Type in the UT and FVT Phases*

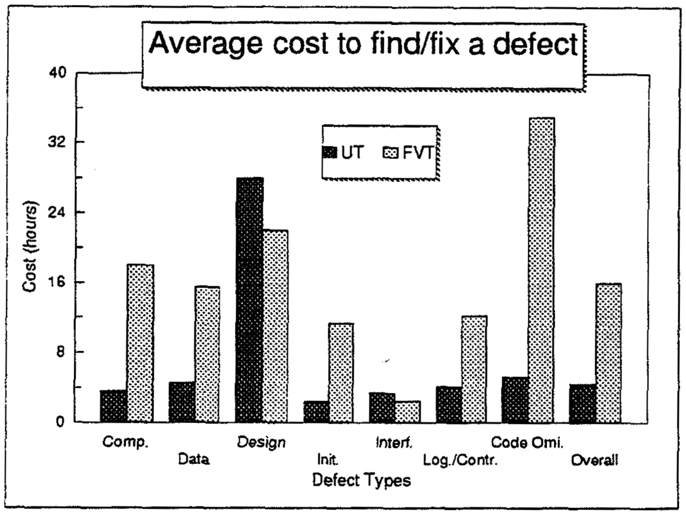


Figure 8: *Comparison of Average Cost (time) per Defect in the UT and FVT Phases*

	Unit Test				Function Test			
	Severity				Severity			
Risk	1	2	3	Total	1	2	3	Total
High	**43%**	0%	3%	**46%**	**7%**	3%	0%	**10%**
Medium	17%	17%	3%	37%	16%	19%	3%	38%
Low	13%	4%	0%	17%	16%	10%	26%	52%
Total	**73%**	21%	6%	100%	**39%**	32%	29%	100%

Table 2: *Comparison of Defect Distribution by Severity and Risk in the UT and FVT Phases*

were focused only on the newly changed or added and directly affected part of the code. However we were not able to filter out the un-changed and not-used functions, in the 50 or so files studied, from being included in the bases for our coverage calculation, because our version of the tool could not yet use granularity lower than source file in selecting monitoring objects. Meanwhile, coverage from test cases generated for related line-items or from applicable ones existed in the regression bucket were not collected (unless they revealed a defect on this line-item), therefore not all coverage from tests performed was reflected in the figures shown, due to the limited resources in doing the study.

This mismatch in data collection and measurement, we believe, is the main reason for the "vertical" growth at the end of FVT phase and the walls on low coverage values shown in the curves presented in this paper.

4.5.4 Why were defects found in FVT after UT was completed

In additional to the quick and easy answer of "unit test did not do a good job", we believe there are more valid points to understand this issue:

- UT and FVT are different in their environments and focuses. Unit testing, by itself, is clearly not adequate. Unit testing is a local (that is, source routine or function level) examination on a small portion of the overall product code and, therefore, is not meant to be complete. The unit test environment itself is much more focused, detailed, and limited; defects arising from interference from other line-items, components or the product's external functions are often not the focus of unit-test. In contrast, it is exactly defects of the latter type that function and integration tests target. The test cases used for integration testing aim at the overall functionality rather than at individual source routines or modules.

- UT and FVT each takes place in a different environment, one on a private build with only changes from its own line-item, the other on the public build where changes made by everyone in the product may affect the test case's execution. Also, as we pointed out earlier, the same test cases, run in a more complicated (integrated) environment may detect new defects that were previously invisible to the private build.

- People who plan and run the test case for the UT and FVT phases are different. The team members that work on the line-item are the only ones to run the UT cases. The FVT cases are usually planned and executed by a separate group. The focus of the FVT group is on testing the external functionalities for the line-item, the integration of the line-items and their co-functionalities. Teams on related line-items and regression testing may also be involved or may have an influence on the FVT of a line-item. The increased perspectives of thinking, the additional involvements, and the combined testing activities of all the above are clearly advantageous in helping identify more defects and contributing to the quality assurance of a line-item.

4.5.5 Why FVT defects are much more expensive to find and fix

We think the cost difference among UT and FVT defects is mainly the results of their differences in three closely related aspects: test environment, test process and test cases.

UT is done on each line-item's private build by the line-item's developers. No test plan or test cases need to be maintained. The test focus is on the correct behavior of each individual routine or module. Test case settings are often simple and manually entered. Frequently the tests are accomplished through the use of a debugger in order to test specific sections or to ensure that certain functions are tested. The developer has full control on the code and the build. Files are often readily compiled with debug options. The developer also has the full control of the test environment. When a problem occurres, the source of the problem is easier to locate, since he is familiar with the code and debugging information is readily available. Changes made for UT defects only affect the private build; so multiple fixes can often be built, tested and verified together.

For FVT, the situation is completely different: test plan and test cases need to be designed, coded and reviewed. The focus of the test is on the overall functionality of the line-item and, on ensuring its correct behavior while it is executed with other functions and in various abnormal conditions. The test cases need to be maintained and automated as much as possible, in order to be merge into the regression bucket at the exit of the FVT of the line-item. For these two reasons, the settings and test scenarios of the test cases are often much more complicated and take much longer to run than the UT cases and frequently, the settings of the whole test suite follow the most complicated test case in the set, for ease of maintenance and automation of the suite. Moreover, FVT is done on a public build, often by a different group. The files in the formal build do not usually contain debugging code, and the public builds are not usually compiled with debugging options. FVT problems are often first looked at by people less familiar with the code than the UT tester was. There are often less debug information available about the problem and, the test environment is often more dynamic and complicated, hence, it is harder to identify the source of the problem than in the UT phase. Often, the problem has to be recreated several times before it can be found; each time, more debugging code are added to the files in order to collect the needed information. Note that even just to add debugging code for problem recreation, since the change needs to be made to the public build, files have to be locked and fastpath tested before add to the build. The cost introduced by these additional file changes and builds, and repeated settings and testings can added up to a significant portion of the overall defect-detection-and-removal cost (time). Further, since the change to fix the defect affects the public build, it has to be handled with more caution. Defect fixes are typically coded, reviewed, unit and fastpath tested, and integrated one by one. Defect information needs to be tracked, and the fix needs to be verified with the original problem scenario before the defect can be closed.

In short, the differences between the UT and FVT testing environments, processes and test cases make every aspect of their defect removal activity, from software building, test setup, test execution, problem recreation, and investigation to coding the fix and verifying the fix, more expensive.

5 What We Learned

This study focused on the early (unit and function) testing phases of our development cycle and on what code coverage could reveal about these phases as well as what light might be shed on the overall testing process. Here are what we have learned from this experience:

- **From an industry perspective**

  There are a few key issues that need to be addressed by the software community:

  - *On value to the industry.*

    For a new technique, tool, or process to be accepted by the industry, it is not sufficient to demonstrate that it is just theoretically sound or has potential benefit. It has to be shown that it can add value to the business, that is, *increase productivity and/or reduce costs or time.*

  - *On integration of testing tools.*

    A tool must be built in such a way that it will seamlessly integrate into the application development environment and be easily incorporated into common development processes. There have been various tools to help software development, understanding, and analysis, but few in the testing area. Testing tools that do exist are often "stand-alone" — they do not interface with other tools, most are not suitable for work in a range of different development tasks, languages, and environments or platforms.

  - *On shifting perspective and attitudes.*

    Test technology and tools have not been receiving the attention and positive attitude they deserve, both from academia and industry. It is an area that, like a gold mine, although challenging to understand and explore, can yield valuable results. As we described in the previous sections, a small improvement in the test process could bring very significant benefit to software product development.

- **On the ATAC tool**

  Out study has shown promising directions on how careful use of code-coverage measurements could aid in unit and function testing (such as showing the insufficiency of test cases and where to add more). It has also identified scalability, robustness, and the user interface as key qualities of software reliability assurance tools (such as ATAC), if they are to be integrated into standard software development processes.

  We found ATAC to be a very promising tool, yet it still is important to have realistic expectations about it. Its current resource consumption and limited scalability greatly restricted the schedule and scope of experiment we would have liked to do with it. Its report granularity has also limited more detailed or accurate analysis of the data in our study; in particular, the way in which code is included (on a per-file basis instead of a use basis) affects the way code coverage is measured. Some initially interested developers were later reluctant to explore the use of the tool more because of its lack of a user friendly interface (this has been addressed in the tool's latest version).

- **On code coverage and test quality**

  The scope and size of our study to date is very limited. There is still much more work to be done to better understand code coverage and its implication for testing. Nevertheless, we found our experiment encouraging in the following ways:

  - There are clear correlations between the growth of defect numbers and the growth of code coverage. We have seen some test cases that increase code coverage without more detecting defects, but all test cases that detected new defect(s) also increased code coverage.
  - Since ATAC can be used to show the code path covered by the current test case and the summary of all previous executed test cases, it is reasonable to expect it to be helpful to developers in
    - indicating areas for additional testing. The ATAC reports have revealed sections of code, that developers thought had been covered, had not been tested at all.
    - generating additional test cases in a manner that provides effective coverage. The coverage information provide by ATAC can help test case developers to direct their focus on the part of the code that had not be tested.
    - identifying the location of defects. This could be very beneficial, even if it was only helpful in some cases, because finding the defects was the most costly portion of work in the test-and-defect-removal process that we studied.
    - eliminating duplicated test cases, as well as the cost to maintain and run them. Such benefit is especially important and significant to the regression test, whose size and execution-cost normally grow continuously with added line-item and from release to release.
  - We have not yet concluded that there is a practical (not 100%) high-end coverage threshold at which to stop testing, although it would not be unreasonable for one to expect that such thresholds exist for the defect/coverage growth curves seen in our previous figures. That is, we expect the existence of a code coverage point beyond which additional test cases would detect no more or limited numbers of defects, but that remains to be proven by future work.

    We believe that such a threshold, if it exists, will vary from one line-item to another and also across testing phases, development environments and software products. However, code-coverage tools can be used to build a test-exit strategy according to these thresholds to dynamically determine the optimized stop point for each line-item, based on its individual situation revealed by its actual testing data, instead of some predetermined arbitrary values.
  - This test-exit problem can be looked at from a different perspective.
    We found that in the FVT phase, although much more costly in detecting and removing each defect in comparing to the UT phase, much less percentage defects detected were critical (high severity or risk). This provides an opportunity to weight defects in a different manner in building an text-exit strategy. That is, instead of using *all*

defect counts when plotting the defect versus coverage curves, we might use the *critical* defect counts. Similarly, code coverage tools can be used to find the optimized test-exit point based on this strategy, for each individual line-item in each testing phase, according to its unique situation as revealed by the dynamic testing data.

- **On potentially beneficial changes to the current testing process.**

  These recommendations remain to be verified by future experiments, but data so far collected and analyzed in our study suggest that:

  - At least half of the defects detected in the FVT phase of the subject line-item could have been found in the UT phase. Among them, there are defects detected by regression tests, related line-items, generic tests, and test cases created specifically for this line-item that did not have dependency on others.

    This indicates there may be advantages in running all available and applicable test cases from all sources during the unit testing phase, and ensure all these test cases are successful prior to the integration of the line-item. Such practice should reveal defects much earlier in the development cycle and thereby reduce the cost (time) spent in subsequent phases.

  - It is also noticed in our study that, the cost-per-defect detected by existing test cases is less than that detected by new test cases specifically generated for the line-item. Therefore, it may be another cost effective practice to first measure, during unit-testing, the coverage of all available and applicable test cases from all sources, then create new test cases only for those sections of code that are not already covered.

  - UT and FVT each have a different focus and tend to detect different types of errors. It is important to realize that even with the improved practices, not all defects can be removed during unit testing. The same test case may reveal additional defects in a changed and/or more complicated (integrated) environment. Therefore it is important to continue running all available test cases from the time a line-item's implementation is completed (in the UT phase, through FVT and SVT) until the product GA.

    Meanwhile, the *critical*-defect-count-versus-code-coverage growth in FVT phase should be used in determining the optimized FVT-exit-point. This can be expected to move the start point for SVT earlier and help ensure better quality software and/or shorten the development cycle.

  - It was not surprising to see, as reported by many other studies[6, 1, 14], that code inspection is the most cost effective defect-removal activity. We recognized however, that code inspection cannot replace testing, because they are activities with different characteristics and each tend to find defects that would be hard for the other to detect.

    What was a little surprising was the fact that even late in the FVT phase, code inspection was still almost as cost-effective as it was in the UT phase. This suggests that it may be a worthwhile quality improving activity to perform additional code inspections in later development cycle (such as FVT and SVT phases), especially if there were not enough time to do a thorough review when the code was implemented.

To maximize the benefit, such review should make use of the information collected in the testing phase (for example test coverage and defect rate) and involve key testers, designer and developers of the line-item, as well as the architecture of the product.

6 Summary and Future Work

This paper has reported on initial work done to evaluate the role of code coverage in providing feedback to developers and testers during the unit-testing and function-testing phases. Based on this study, we can draw several conclusions:

- Many high-risk/high-severity defects were found in unit testing, while most defects detected in the function test phase were lower-risk/lower-severity ones. In contrast, the cost of defect-removal in function-testing phase was significantly higher, in fact, three to four times the average cost per defect removed in the unit-test phase.
- Even though function testing increased the code coverage only slightly, the relative number of defects found was significant. This is not to suggest that every 1% or 2% increase in code coverage yields significant numbers of defects. Rather, we suspect that the defects found during the function-test phase were significant. Moreover, we believe that the way in which the metrics are generated may not adequately reflect the true code covered, which we suspect is much higher. This hypothesis needs to be verified and is one objective of future work.
- Our results lead us to conclude that the strategy we used in determining when to move to function and system testing may not be the best. It may be better to move many activities currently performed only in the function-testing phase to the unit-testing phase, and move to system test when there are only low-risk/low-severity function-test defects left. That would better utilize the overall time available.

Although the results were preliminary, it is encouraging to note the additional insight that code coverage tools can provide to the testing process, when compared to traditional measurements, such as the number of test cases run or the amount of test execution time. Using the the traditional measurements, we observe the external attributes and parameters of the test process. With the code-coverage method, we evaluate internal attributes. This new perspective can be used, we believe, to increase the efficiency and effectiveness of early software testing to assure software reliability.

We also believe that, just as code inspection, unit test and function test is complementary to each other in the software defect removal process, the code coverage approach should only be considered a complement, not a replacement, of the traditional approaches in measuring the test quality and reliability growth of a software.

This work is a part of an ongoing project carried out in the IBM SWS Toronto Lab, led by the Lab's Centre for Advanced Studies and in conjunction with Bellcore and several universities. The project will continue to explore the utility of such code coverage techniques in the remainder of the software development cycle – the system-verification-and-acceptance-testing phase, and to analyze the function-and-integration-testing phase based on data from the entire product.

Acknowledgment

Thanks go to *Soal London* and *Bob Horgan* from **Bellcore**, *Aditya Mathur*, *Praerit Garg* and *Saileshwar Krishnamurthy* from **Purdue University**, *Pei Hsia* and *Frederick Sheldon* from **University of Texas**, *Mladen A. Vouk*, *Kuo Chang Tai* and *Amit M. Paradkar* from **State University of North Carolana**, *Phil Ford* and *Karen Bennet* from **IBM SWS Toronto CAS**, and people in the **IBM SWS Toronto Lab development and test groups**, especially *Colin L. Mackenzie*, *Doug Nybida*, *Dave Godwin*, *Kaarel Truuvert*, *Joe Palma*, *Aleem Rajpar*, *Roger Zheng*, *Eric Labadie*, *Hebert Pereyra Patrick MacDonald*, *Brett Macintyre*, *Harry Chow* and *Farzad Amin.*

Participants of all these partners have contributed in one form or another to the project, the vision, the approach and the technics. Reported in this paper are the work of all those participants.

Special thanks go to **Dennis Bockus** for his contribution to the preparation of the paper within a very brief period of time.

Thanks also go to many others who contributed to the project. It is, unfortunately, not possible to list all the names here. We hope they understand.

References

[1] M. J. Bassman, F. McGarry, and R. Pajerski. Software Measurement Guidebook. Technical Report SEL-94-002, NASA, July 1994.

[2] Y. Biggerstaff and C. Richter. Reusability Framework, Assessment and Directions. *IEEE Software*, SE-2(4):41–49, 1987.

[3] M. Cheek. Cost-conscious Developers Looking to Computer-aided Testing. *IEEE Software*, 11(4):108–109, 1994.

[4] W. Cornelissen. How to Make Intuitive Testing More Systematic. *IEEE Software*, 12(5):87–89, 1995.

[5] J. Dobbins. *Handbook of Software Quality Assurance*, chapter "Measurement for Software Reliability". Van Nostrand Reinhold, 1986.

[6] M. E. Fagin. Design and Code Inspections to Reduce Errors. *IBM Systems Journal*, 15(3):216–245, 1976.

[7] J. R. Horgan and S. London. A Data Flow Coverage Testing Tool for C. In *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*, pages 2–10, 1992.

[8] J. R. Horgan, S. London, and M. R. Lyn. Software Quality with Testing Coverage Measures. *IEEE Computer*, 27(9):60–70, 1994.

[9] W. S. Humphrey. *Managing the Software Process*, chapter "Defining the Software Process", page 251. Addison-Wesley Publishing Company, 1989.

[10] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4), 1976.

[11] K. Moller. *Software Quality and Reliability*, chapter "Increasing Software Quality by Objectives and Residual Fault Prognosis". Chapman and Hall, 1991.

[12] J. Musa, A. Iannino, and K. Okumoto. *Software Reliability.* McGraw-Hill Publishing, 1990.

[13] A. K. Onoma and T. Yamaura. Practical Steps toward Quality Development. *IEEE Software*, 12(5):68–77, 1995.

[14] A. Porter, H. Siy, C. Toman, and L. Votta. An Experiment to Assess the Cost-benefit of Code Inspections in Large Scale Software Development. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 92–103, 1995.

[15] R. Poston. Testing Tools Combine Best of New and Old. *IEEE Software*, 12(2):122–127, 1995.

[16] S. T. Redwine. Two Industrial Testing Cultures Meet at STAR'94. *IEEE Software*, 11(4):108–109, 1994.

[17] T. C. Royer. *Software Testing Management: Life on the Critical Path.* Prentice-Hall Publishing, 1994.

[18] J. Slonim, M. Bauer, P. Larson, J. Schwarz, C. Butler, E. Buss, and D. Sabbah. The Centre for Advanced Studies: A Model for Applied Research and Development. *IBM Systems Journal*, 33(3), 1994.

[19] W. Smith. Making War on Defects. *IEEE Spectrum*, 30(9):43–50, 1993.

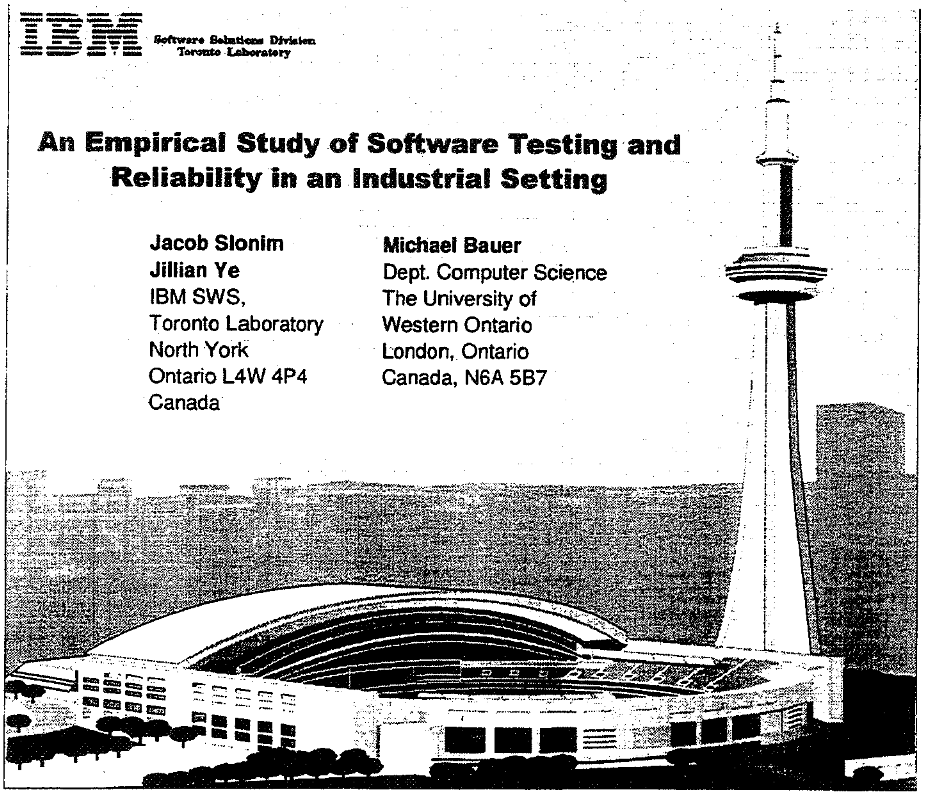
[20] R. Tausworthe. Experience More Reliable than Theory at 5th ISSRE. *IEEE Software*, 12(2):110–112, 1995.

[21] J. Tian, P. Lu, and J. Palma. Test-execution-based Reliability Measurement and Modelling for Large Commercial Software. *IEEE Transactions on Software Engineering*, 21(5):405–414, 1995.

[22] J. Tian and M. Zelkowitz. Complexity Measure Evaluation and Selection. *IEEE Transactions on Software Engineering*, 21(8):641–650, 1995.

[23] E. Weyuker. Using the Consequence of Failures for Testing and Reliability Assessment. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–91, 1995.

[24] C. Wohlin and U. Kormer. Software Faults: Spreading, Detection and Costs. *Software Engineering Journal*, January:33–42, 1990.

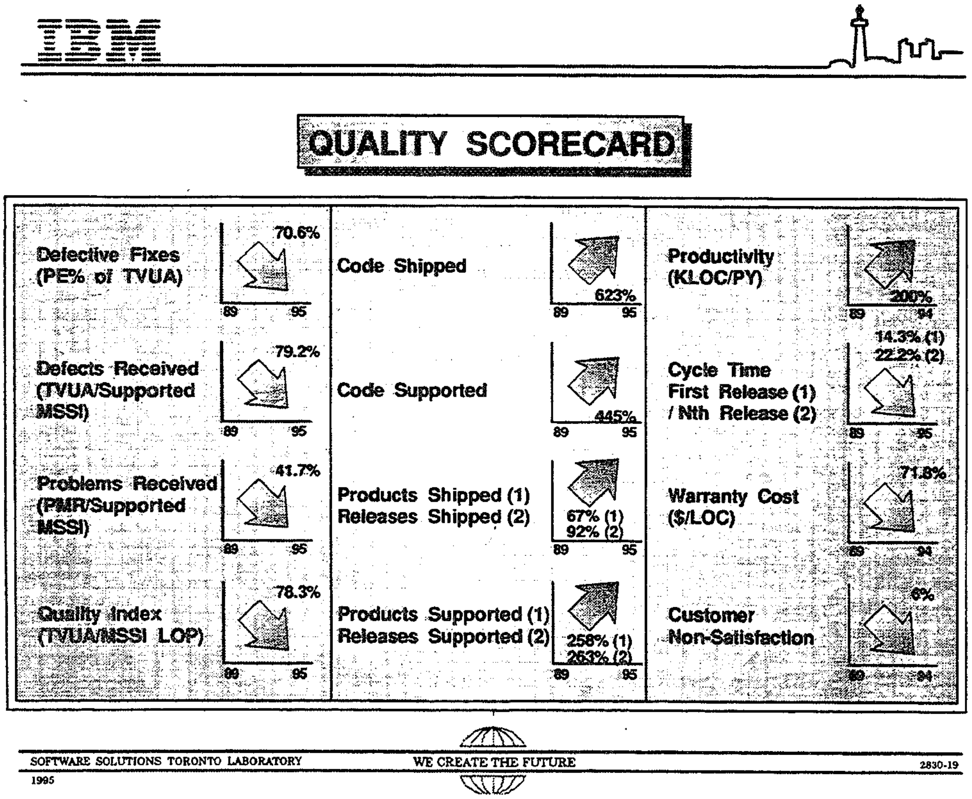


IBM Software Solutions Division Toronto Laboratory

An Empirical Study of Software Testing and Reliability in an Industrial Setting

Jacob Slonim
Jillian Ye
IBM SWS,
Toronto Laboratory
North York
Ontario L4W 4P4
Canada

Michael Bauer
Dept. Computer Science
The University of
Western Ontario
London, Ontario
Canada, N6A 5B7



IBM

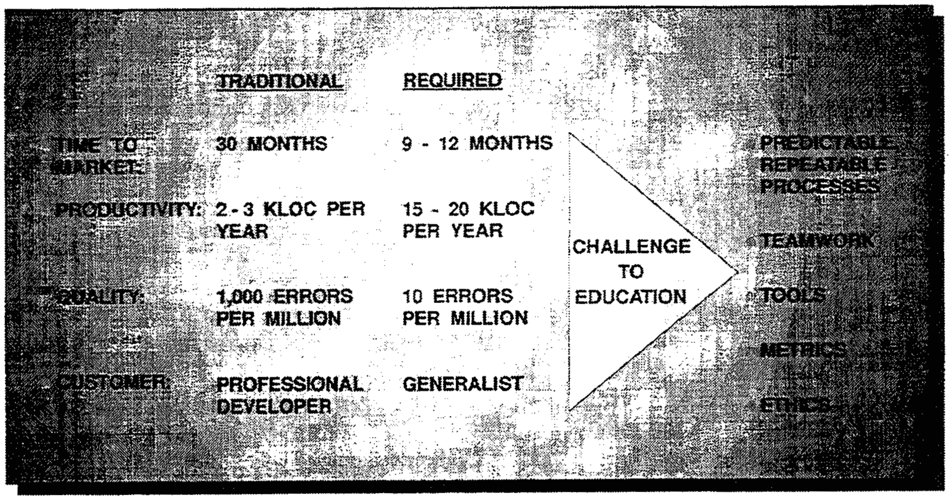
QUALITY SCORECARD

Defective Fixes (PE% of TVUA)	70.6%	89	95
Defects Received (TVUA/Supported MSSI)	79.2%	89	95
Problems Received (PMR/Supported MSSI)	41.7%	89	95
Quality Index (TVUA/MSSI LOP)	78.3%	89	95
Code Shipped	623%	89	95
Code Supported	445%	89	95
Products Shipped (1) Releases Shipped (2)	67% (1) 92% (2)	89	95
Products Supported (1) Releases Supported (2)	258% (1) 263% (2)	89	95
Productivity (KLOC/PY)	200%	89	94
Cycle Time First Release (1) / Nth Release (2)	14.3% (1) 22.2% (2)	89	95
Warranty Cost ($/LOC)	71.8%	89	94
Customer Non-Satisfaction	6%	89	94

SOFTWARE SOLUTIONS TORONTO LABORATORY
1995
WE CREATE THE FUTURE
2830-19

IBM CAS

SOFTWARE PROFESSIONAL



	TRADITIONAL	REQUIRED
TIME TO MARKET:	30 MONTHS	9 - 12 MONTHS
PRODUCTIVITY:	2-3 KLOC PER YEAR	15 - 20 KLOC PER YEAR
QUALITY:	1,000 ERRORS PER MILLION	10 ERRORS PER MILLION
CUSTOMER:	PROFESSIONAL DEVELOPER	GENERALIST

CHALLENGE TO EDUCATION

- PREDICTABLE REPEATABLE PROCESSES
- TEAMWORK
- TOOLS
- METRICS
- ETHICS

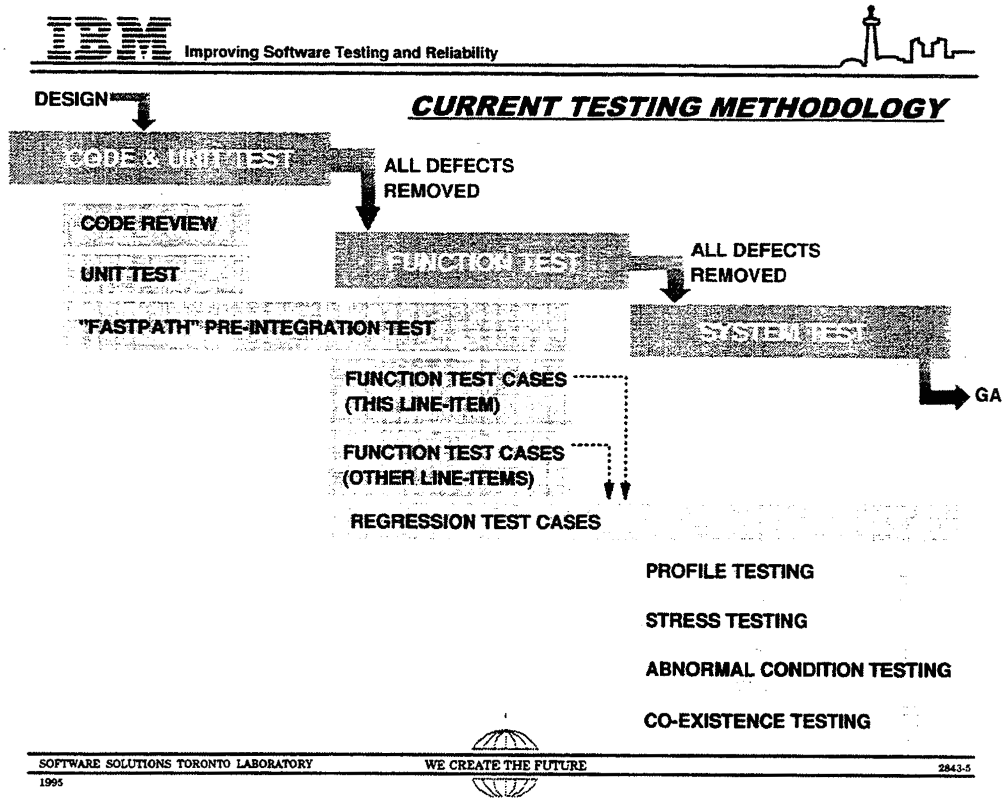
1995 SOFTWARE SOLUTIONS TORONTO LABORATORY 2521-11

IBM Improving Software Testing and Reliability

CHALLENGES

- Lack of quantitative measurement of testing effort
- Lack of feedback to direct effort for improvement of software reliability / defect detection
- Cost of Regression test grow continuously with the volume of test buckets
- Increase automation

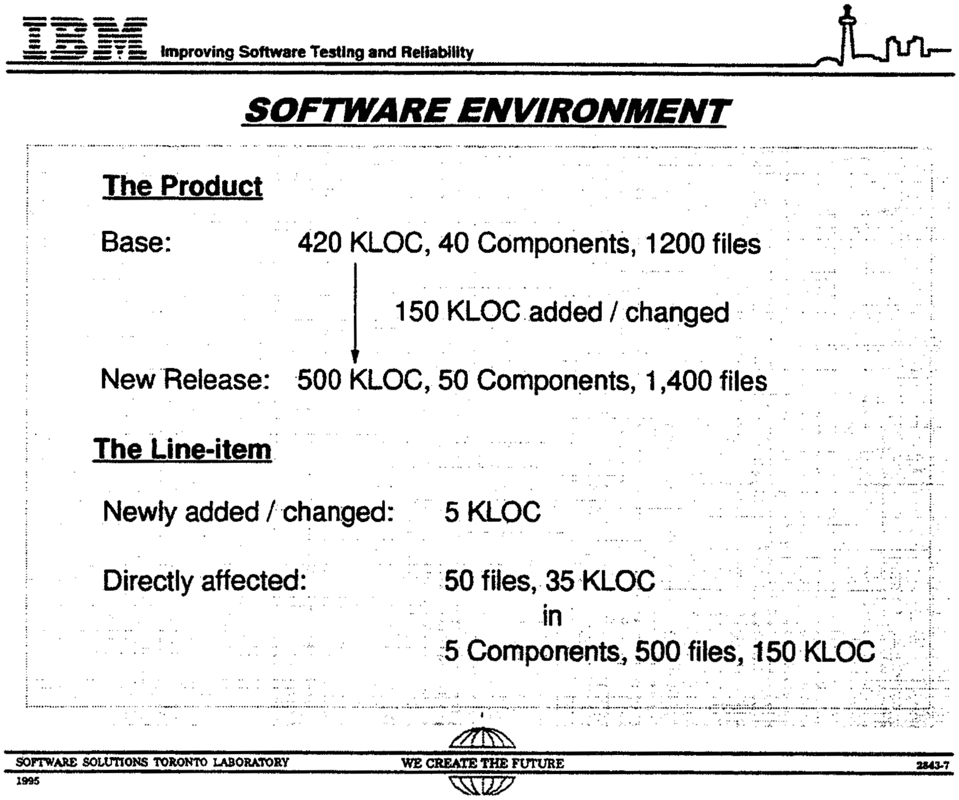
SOFTWARE SOLUTIONS TORONTO LABORATORY WE CREATE THE FUTURE 2843-2
1995



IBM Improving Software Testing and Reliability

CURRENT TESTING METHODOLOGY

SOFTWARE SOLUTIONS TORONTO LABORATORY 1995 WE CREATE THE FUTURE 2843-5



IBM Improving Software Testing and Reliability

SOFTWARE ENVIRONMENT

The Product

Base: 420 KLOC, 40 Components, 1200 files

150 KLOC added / changed

New Release: 500 KLOC, 50 Components, 1,400 files

The Line-item

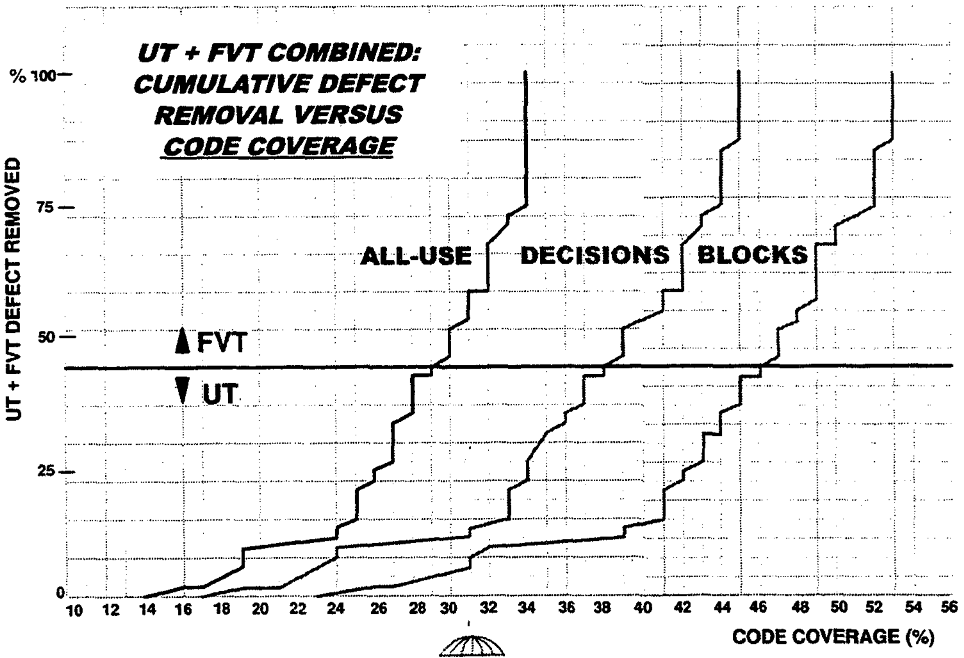
Newly added / changed: 5 KLOC

Directly affected: 50 files, 35 KLOC
in
5 Components, 500 files, 150 KLOC

SOFTWARE SOLUTIONS TORONTO LABORATORY 1995 WE CREATE THE FUTURE 2843-7

NOTES ON DATA COLLECTION

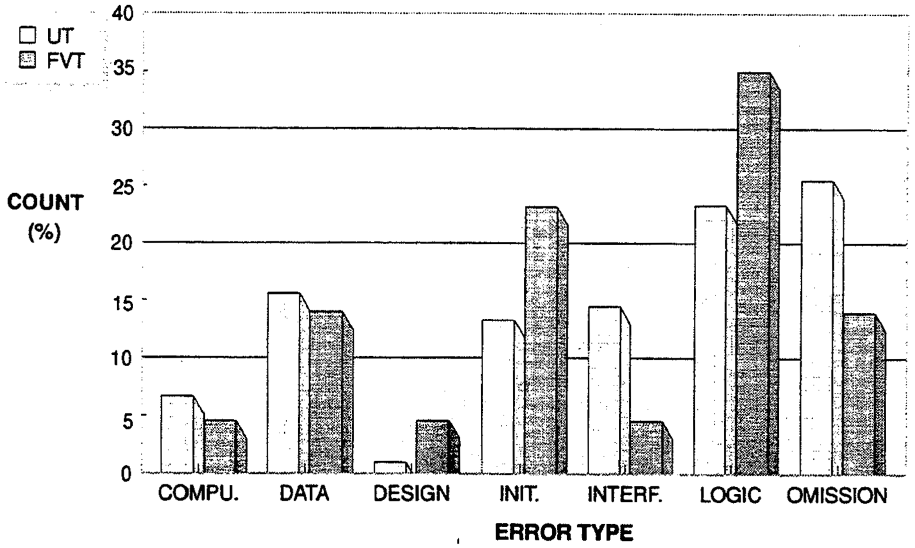
- Focused on **one** line-item
- All UT and FVT defects are counted
- Coverage measurements are done on **all** modules (files) changed and added by the line-item
  (Many of these files contain functions not needed by the line-item)
- UT: only test cases that detected defects are counted
- FVT: - all test cases specifically generated for this line-item are counted
  - Regression test cases and other line-items' test cases each is counted only if it found a defect for this line item



UT + FVT COMBINED: CUMULATIVE DEFECT REMOVAL VERSUS CODE COVERAGE

IBM Improving Software Testing and Reliability

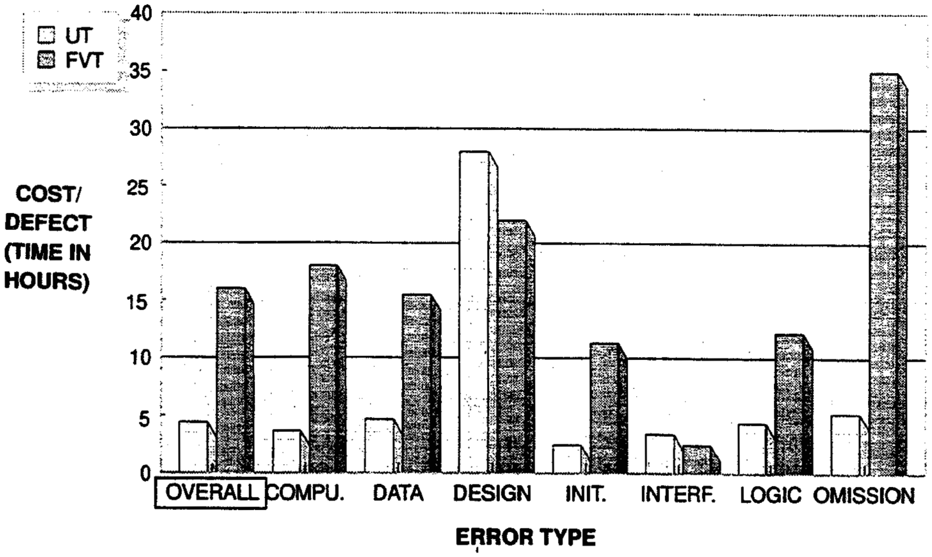
DEFECT COUNT DISTRIBUTION BY ERROR TYPE



SOFTWARE SOLUTIONS TORONTO LABORATORY 1995 WE CREATE THE FUTURE 2843-10

IBM Improving Software Testing and Reliability

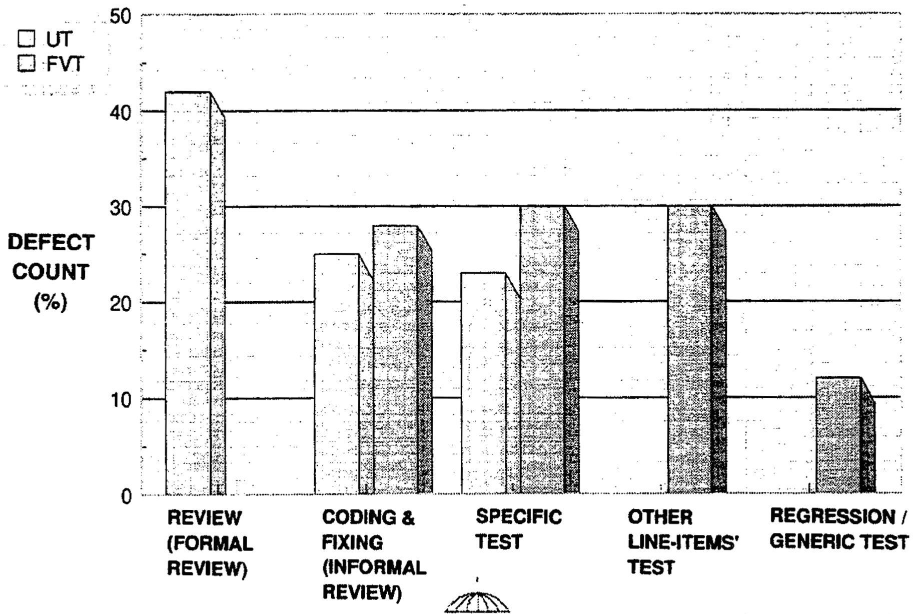
COST PER DEFECT BY ERROR TYPE



SOFTWARE SOLUTIONS TORONTO LABORATORY 1995 WE CREATE THE FUTURE 2843-12

IBM Improving Software Testing and Reliability

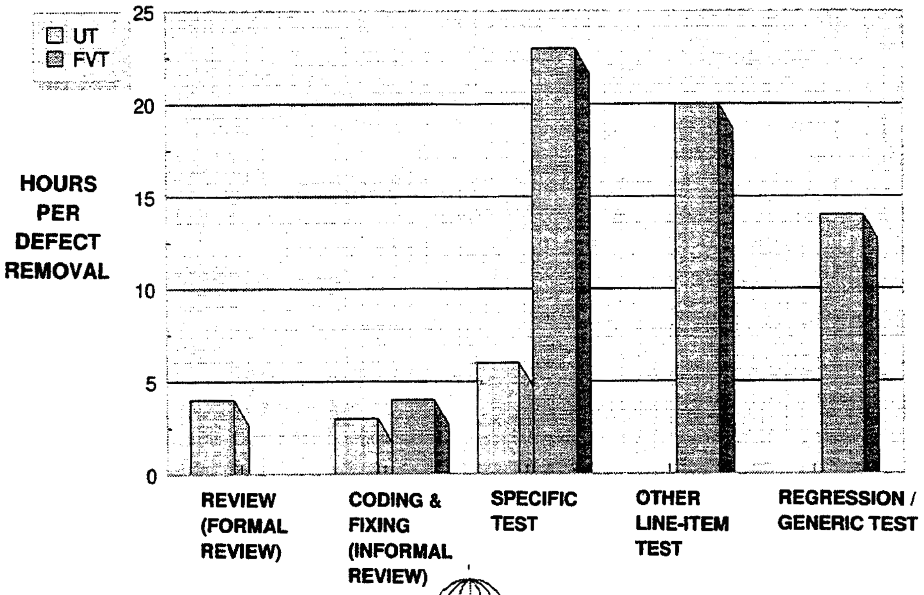
DEFECT DISTRIBUTION BY SOURCE OF DETECTION



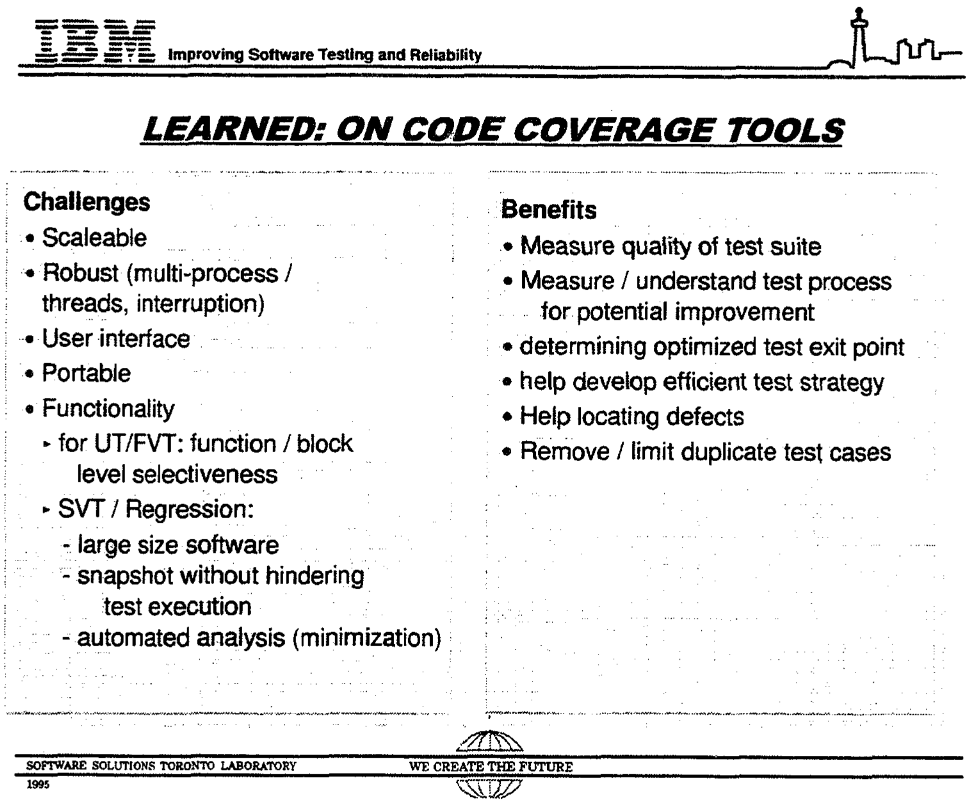
SOFTWARE SOLUTIONS TORONTO LABORATORY 1995 WE CREATE THE FUTURE 2843-18

IBM Improving Software Testing and Reliability

COST PER DEFECT BY SOURCE OF DETECTION



SOFTWARE SOLUTIONS TORONTO LABORATORY 1995 WE CREATE THE FUTURE 2843-19



Improving Software Testing and Reliability

LEARNED: ON CODE COVERAGE TOOLS

Challenges

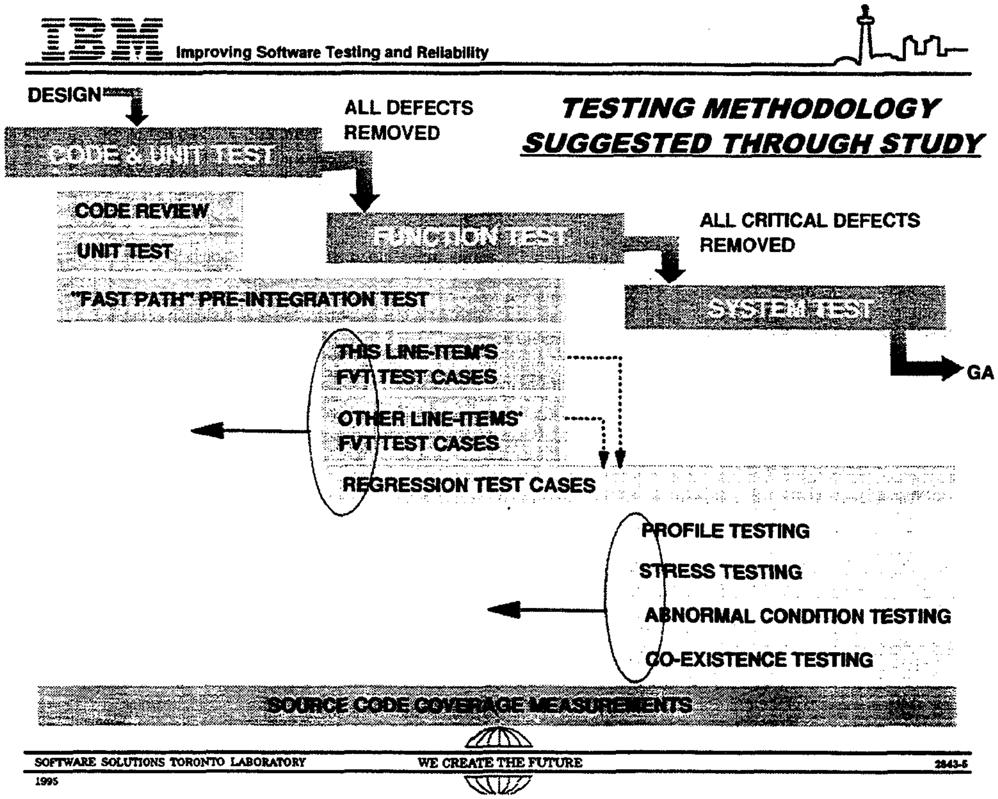
- Scaleable
- Robust (multi-process / threads, interruption)
- User interface
- Portable
- Functionality
  - for UT/FVT: function / block level selectiveness
  - SVT / Regression:
    - large size software
    - snapshot without hindering test execution
    - automated analysis (minimization)

Benefits

- Measure quality of test suite
- Measure / understand test process for potential improvement
- determining optimized test exit point
- help develop efficient test strategy
- Help locating defects
- Remove / limit duplicate test cases

SOFTWARE SOLUTIONS TORONTO LABORATORY WE CREATE THE FUTURE

1995



Improving Software Testing and Reliability

TESTING METHODOLOGY SUGGESTED THROUGH STUDY

DESIGN
CODE & UNIT TEST
ALL DEFECTS REMOVED
CODE REVIEW
UNIT TEST
FUNCTION TEST
ALL CRITICAL DEFECTS REMOVED
"FAST PATH" PRE-INTEGRATION TEST
SYSTEM TEST
GA
THIS LINE-ITEM'S FVT TEST CASES
OTHER LINE-ITEMS' FVT TEST CASES
REGRESSION TEST CASES
PROFILE TESTING
STRESS TESTING
ABNORMAL CONDITION TESTING
CO-EXISTENCE TESTING
SOURCE CODE COVERAGE MEASUREMENTS

SOFTWARE SOLUTIONS TORONTO LABORATORY WE CREATE THE FUTURE 2843-5

1995

Software-Reliability-Engineered Testing

S5-61

41560/

16 P.

360691

John D. Musa

AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

Software-reliability-engineered testing (SRET) is testing that is designed and guided by reliability objectives and expected field usage and criticality. It generally includes feature, load, and regression testing.

SRET is unique in helping testers insure the necessary reliability of a system in minimum delivery time and cost. It increases tester efficiency and reduces the risk of angry customers, as compared to nonengineered testing. It is a standard, proven practice.

We can cost-effectively apply SRET to every software-based product and to the frequently used members of component libraries. It involves activities that extend over all releases of a system and all life-cycle phases, but there is special focus on system test.

We define **reliability** *for software [1] as the probability of execution without failure for some specified interval, generally called the* **mission time**. *Thus we use a definition that is compatible with that used for hardware reliability, though the mechanisms of failure may be different. This is because we want to be able to work with software-based systems that are composed of both software and hardware components - note that in practice there are no pure software systems.*

This paper presents the benefits of SRET as contrasted with nonengineered testing and it describes the practice of SRET in some detail, based on extensive experience, particularly in AT&T.

A Valuable Standard Practice

Software-reliability-engineered testing is currently being practiced in a substantial number of projects in AT&T. It is based on and included in the AT&T "Best Current Practice" of Software Reliability Engineering (SRE) that was approved in May 1991. Qualification as a best current practice requires use on several projects (typically 8 to 10) with documented strong benefit/cost ratios and probing review by two boards of high-level managers; 1 of 6 proposals was approved in 1991.

In one large AT&T software development organization (Operations Technology Center of Network Services Division) with over 1500 developers and some 70 projects, 30% used SRE as of April 1995 and it has continued to increase since then. In this organization, it is part of the standard software development process and is currently undergoing ISO certification. It is interesting to note that this organization, which has the highest percentage use of SRE in AT&T, is the primary development organization for the AT&T business unit that won the Malcolm Baldrige National Quality Award in 1994. In addition, 4 of the first 5 software winners of the AT&T Bell Laboratories President's Quality Award applied SRE.

The International Definity® project represents one application of SRE. They applied it along with some related technologies. In comparison with a previous release that did not use these technologies, they increased customer satisfaction significantly (sales increased by a factor of 10), with reliability increasing by a factor of 10. There were reductions of a factor of 2 in system test interval and system test costs, 30% in total project development interval, and a factor of 10 in program maintenance costs.

Although AT&T has perhaps been a leader in the use of SRET, many other organizations have applied it or the somewhat more inclusive software reliability engineering practice. An AIAA standard was approved in 1993 and IEEE standards are under development. McGraw-Hill and the IEEE Computer Society Press are publishing a handbook [2].

What Should We Test?

We will, of course, want to test the actual product software-based system we are developing. However, we will also want to identify as systems to be tested those major components of unknown reliability whose operational profiles (to be defined) are known or easily determined and whose *execution time* (the actual time expended by the processor in executing the software in question) or an approximation thereof is readily measurable. If components (not necessarily major) have the foregoing characteristics and we expect to reuse them extensively, they can profitably be tested by themselves. If the software-based product interacts strongly with other software-based systems, we may want to test a supersystem that represents these systems functioning together.

There are two types of software-reliability-engineered testing, development testing and certification testing. The main objective of development testing is to find and remove faults. During development testing, you use SRET to estimate and track *failure intensity* (failures per unit execution time). Testers apply the failure intensity information to determine any corrective actions that might need to be taken and to guide release. The release decisions include release from system test to beta test and release from beta test to general availability. You typically use development testing for software you "developed" in your own organization.

Certification does not involve debugging. There is no attempt to "resolve" failures you identify by determining the faults that are causing them and removing the faults. With certification testing you make a binary decision: accept the software, or reject the software and return it to its supplier for rework. You typically use certification for software you acquire. *Acquired software* includes "off the shelf" or packaged software, software that is reused, and software that is developed by an organization other than the product development organization. There appears to be great potential in the application of software reliability engineering to certify object libraries needed for object-oriented development. In fact, the further growth and use of object-oriented development may depend on this marriage of technologies: object-oriented concepts have made better modularization possible, but the promise and benefits of reuse are not being fully realized because developers (probably rightly!) have enormous resistance to using objects whose reliability they cannot vouch for.

FONE FOLLOWER

Let's consider an illustration that we will apply throughout this paper to help make the process of software-reliability-engineered testing more concrete. We draw the illustration from an actual project, with the details somewhat modified for the purposes of simplicity and protecting any proprietary information.

FONE FOLLOWER is a system that enables you to make your telephone calls "follow" you anywhere in the world according to a program that you enter. You as user dial into a voice response system and enter the telephone numbers (they can be for cellular phones) at which you plan to be as a function of time. Most of these phone number entries are made between 7 and 9 AM each day.

Calls that would normally be routed to your telephone are sent to FONE FOLLOWER. It forwards them in accordance with the program you entered. If there is no response, you are paged if you have pager service. If there is still no response or if you don't have pager service, the calls are forwarded to your voice mail.

System engineering has decided that FONE FOLLOWER will use a vendor-supplied operating system. The reliability of the operating system is not known, but we can easily specify an operational profile for it, because we know how FONE FOLLOWER will use it. FONE FOLLOWER does not interact substantially with other systems in the telecommunication network.

We can conclude from the foregoing information that we will identify two systems for software-reliability-engineered testing. We will development test the FONE FOLLOWER product and certify the operating system.

SRET Process

The SRET process consists of five principal activities. These are shown in Figure 1, along with the project phases in which you customarily perform them. Note the "execute tests" and "interpret failure data" occur simultaneously and are closely linked, with the relative emphasis on interpretation increasing with time.

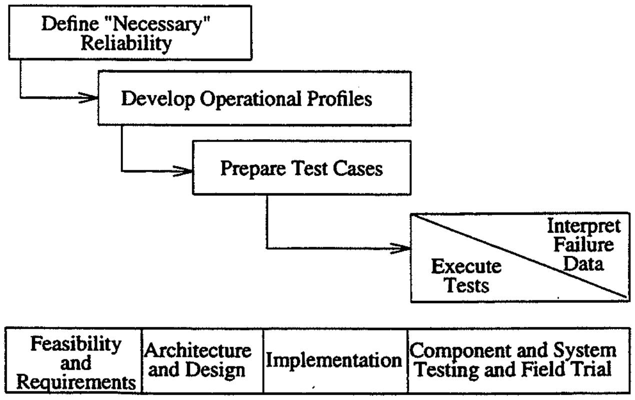

Define "Necessary" Reliability
Develop Operational Profiles
Prepare Test Cases
Interpret Failure Data
Execute Tests
Feasibility and Requirements
Architecture and Design
Implementation
Component and System Testing and Field Trial

Figure 1. SRET Process Diagram

Testers conduct the first two activities, "define necessary reliability" and "develop operational profiles," in partnership with system engineers. We originally thought that these activities should be assigned solely to system engineers and system architects. However, this did not work well in practice. Testers depend on these activities and are hence more strongly motivated than system engineers and system architects to insure their successful completion. We found that the problem was resolved when we made testers part of the system engineering and system architecture team. This approach also had unexpected side benefits. Testers had much more contact with product users, which was very valuable in knowing what system behavior would be unacceptable and how unacceptable it would be, and in understanding how users would employ the product. System engineers and system architects obtained a greater appreciation of testing and of where requirements and design needed to be made less ambiguous and more precise, so that test planning and test case and test procedure design could proceed. System testers made valuable contributions to architecture reviews, often pointing out important capabilities that were missing.

Define "Necessary" Reliability

In order to define the necessary reliability for each system we are analyzing for our product development, we must:

1. determine which operational modes need reliability verification,
2. define "failure" with severity classes, and
3. set failure intensity objectives (per severity class).

Failure intensity, incidentally, is an alternative way of expressing software reliability. It is defined as failures per unit execution time. For example, failure intensity might be 6 failures/1000 CPU hr.

An *operational mode* is a distinct pattern of system usage that is likely to stimulate different failures or rarely-occurring failures with critical impact. Thus, an operational mode needs separate testing. Some of the factors that may yield different operational modes are day of week or time of day (prime hours vs off hours), system maturity (new system vs mature system), special conditions such as system being partially operational or in overload, and rare critical events. Division into operational modes is based on engineering judgment: more operational modes can increase the realism of test but they also increase the effort and cost of selecting test cases and performing system test.

We will select 4 operational modes for FONE FOLLOWER, based on 2 traffic level variables, phone number entries traffic and calls traffic. The 4 modes are then:

MODE	ENTRIES TRAFFIC LEVEL	CALLS TRAFFIC LEVEL
1	Average	Average
2	Average	Peak
3	Peak	Average
4	Peak	Peak

Let's now consider how we define "failure" and the different severity classes. A *failure* is a departure of program behavior in execution from user requirements; it is a user-oriented concept. A *fault* is the defect in the program that causes the failure when executed, a developer-oriented concept. We point this out to emphasize the fact that defining failures implies establishing negative requirements on program behavior, as desired by users. The definition process consists of outlining these requirements in a project-specific fashion for each severity class.

A *severity class* is a set of failures which share the same degree of impact on users. Common classification criteria include human life impact, cost impact, and service impact. In general, classes are widely separated in impact because it isn't possible to estimate impact with high accuracy.

For FONE FOLLOWER, we will use service impact as the severity class classification criterion. Defining "failure" specifically for this product in terms of different severities, we have:

SEVERITY CLASS	FAILURE DEFINITION
1	Failure that prevents calls from being forwarded
2	Failure that prevents phone number entry
3	Failure that makes system administration more difficult although possible through alternate means: for example, can't add or delete users from graphical user interface
4	Failure of function that is deferrable, such as preventive maintenance

In applying SRET, we set system failure intensity objectives (FIOs) based on analysis of specific user needs, the existing system reliability and the level of user satisfaction with it, and the capabilities of competing systems. The objectives may be different for different operational modes and different severity classes. Also, there may be different objectives for the end of system test and the end of beta test.

We determine the failure intensities in clock hours of the hardware and the acquired software components in the system (these will be certified at acceptance of delivery). Then we subtract these from the system failure intensity objectives in clock hours to find the failure intensity objectives required for the developed software. The results are converted into failure intensity objectives per CPU hour.

Developing the Operational Profile

An *operation* is an externally- and independently-initiated complete task performed by a system. It is a logical rather than a physical concept, in that an operation can be executed over several machines and it can be executed in noncontiguous time segments. An operation can be initiated by a user, another system, or the system's own controller. Some examples of operations are a command activated by a user, a transaction sent for processing from another system, a response to an event occurring in an external system, and a routine housekeeping task activated by your own system controller.

The *operational profile* is simply the set of operations and their probabilities of occurrence.

For FONE FOLLOWER, in the operational mode defined by average phone entries traffic level, average calls forwarded traffic level, we have 5000 operations/clock hr. Note the following segment of the operational profile, and how we can obtain it from the occurrence rates of individual operations:

OPERATION	OPER/ CLOCK HR	PROBABILITY
Connect call	2000	0.40
Connect to voice mail	1600	0.32
• • •		
	5000	1

To develop an operational profile, you:

1. Identify the initiators of operations,
2. Enumerate the operations that are produced by each initiator,
3. Determine the occurrence rates (per CPU hour) of the operations, and
4. Determine the occurrence probabilities by dividing the occurrence rates by total operation occurrence rates.

The initiators of operations are most commonly users of the systems, but they can also include external systems and the system's own controller. To identify the users, you first determine the expected customer types for the system, based on such information as the system business case and marketing data for related systems. You then analyze the customer types for the expected user types or sets of users who will tend to use the system in the same way. User types are often highly correlated with job roles.

In order to list the operations for each initiator, you should primarily consult the system requirements. However, other useful sources include work process flow diagrams for various job roles, draft user manuals, prototypes, and previous versions of the system. Direct discussions with "typical" expected users are usually highly enlightening.

Many first-time users of SRET expect that determining occurrence rates for operations will be very difficult; our experience generally indicates much less difficulty than expected. Frequently, field data already exists for the same or similar systems, perhaps previous versions. If not, you can often collect it. If the operations are event driven, you can often simulate the environment that determines event frequency. Finally, even if there is no direct data, there is usually some related information that lets you make reasonable estimates.

An important task associated with developing the operational profile is the identification of critical operations, based on the safety or value they add when satisfactorily executed, or the risk to human life, cost, or reduction in capability resulting from failure.

Prepare Test Cases

Testing consists principally of feature testing, regression testing, and load testing. In feature testing, test runs are executed independently of each other, with the data base reinitialized each time, such that they do not interact. In regression testing, feature test runs are repeated after changes have been made to the system to see if any failures occur that are the result of faults spawned in the process of change. In load testing, large members of test runs are executed in the context of a test procedure that realistically models load and hence includes the interactions that can occur among test runs, both directly and through the slowly corrupting data base.

When we specify the test cases we plan to prepare, we should follow the general principle that we never duplicate the same test run (instance of an operation, with all conditions specified), because this wastes testing resources without yielding any new information about the system. By "duplicate," we mean "execute with exactly the same values for all input variables." An input variable is any variable that exists external to a system and affects its operation. You can make exceptions to this rule when you need to gather more data from a run, verify that a failing run now operates successfully, or conduct a regression test.

When we specify a test case, we also specify the test run except for the test procedure in which it executes. Each test procedure will have somewhat different conditions. Hence the same test case can run in different test procedures, yielding different test runs. Consequently, before we specify the test cases we will use for our system, we will identify the test procedures and use them to reduce the number of test cases we need. Note that if we run a test case without a test procedure, we are in effect doing feature testing, so specification of test cases takes care of all three types of testing: feature, regression, and load.

The relationships among operational modes, test procedures, operations, runs and test cases are shown in Figure 2.

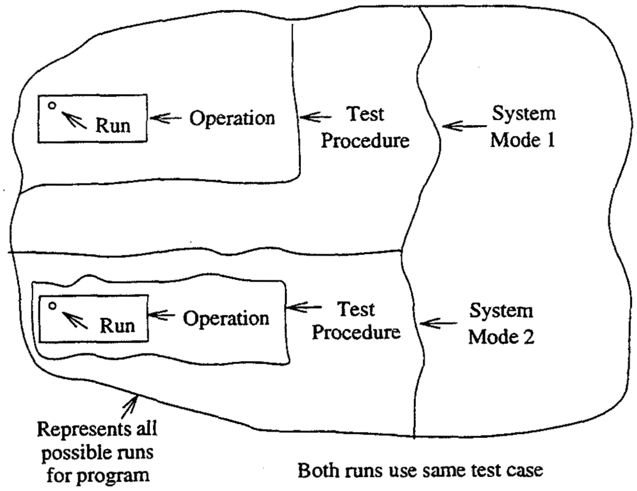


Figure 2. Test Cases, Test Procedures, and Test Runs

The procedure for preparing test cases involves:

1. estimating the number of test cases needed,
2. specifying the test cases, and
3. preparing the test case and test procedure scripts.

Test case specification generally proceeds in two steps, selection of the operation and selection of the run. The operation is selected with selection probability equal to its occurrence probability in the operational profile. The run is selected with equal probability of selection from among all the possible runs of the operation.

It is not necessary that selection be random, although it is desirable because it helps avoid unconscious bias. Because of the large number of test cases expected, it is recommended that the selection be automated wherever possible.

Once the test cases are selected, the test scripts for them must be prepared. There are many software support tools that can help with this task. In addition, you should provide for recording operations executed, so that you collect use data for comparing the operational profile used in test with that expected in the field. In reality, recording of operations should not be a feature of just the test platform but an integral part of the system itself, so that extensive field data can be collected, both to evaluate the current system and to provide a base for engineering future systems.

Execute Tests

Each operational mode, and in fact each test procedure is executed separately. Execution of tests will be facilitated if you use a test management system to help you set up, execute, and clean up sets of tests, with capture of input and output.

We identify failures, determine when they occurred, and establish the severity of their impact. Initially, you look for deviations from behavior, which can be intermediate or user-affecting (only the latter represent actual failures). There are many standard types of deviations that can be detected with generic tools: interprocess communication failures, illegal memory references, deviant return code values, memory or other resource leaks, deadlocks, resource threshold overruns, process crashes, etc. In addition, assertions can be manually inserted in the code to set flags that permit programmer-defined deviations to be detected by generic tools. However, some degree of manual inspection of test results will probably be necessary to identify failures not amenable to automatic detection and to sort out those deviations that are true failures, unless you can demonstrate that the ratio of failures to deviations is essentially constant. In the latter case, you can use deviation data in place of failure data, adjusting by the known ratio.

Interpret Failure Data

You will interpret failure data differently for development testing and certification testing.

During system test of developed software, we make periodic estimates of failure intensity based on failure data.

Failure intensity estimates are typically made from failure times or failures per time period, using reliability estimation programs such as CASRE [2] that are based on software reliability models and statistical inference.

We look at the trend of failure intensity in time (see Figure 3). The center plot represents the most likely (maximum likelihood) estimate and the other two plots the upper and lower 75% confidence bounds. Upward discontinuities in failure intensity commonly indicate either system evolution or a change in the operational profile that represents the runs currently being made. System evolution can be an indication of poor change control. An operational profile change may indicate nonstationary test selection or an inaccurate operational profile. In either case, corrective actions are necessary if you are to have a quality test effort that you can rely on.

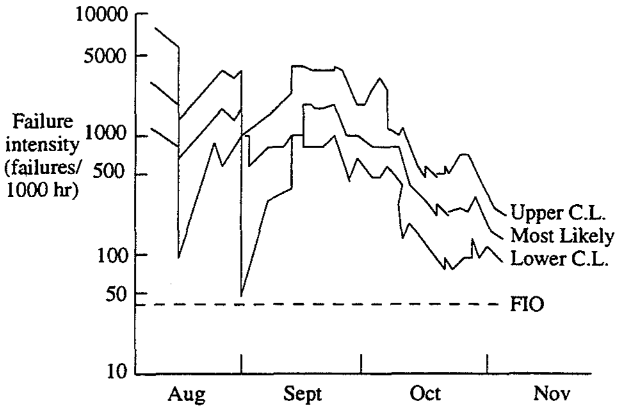


Figure 3. Example Failure Intensity Trend

We compare failure intensities with their corresponding failure intensity objectives (remember that these can be multiple; for example, involving different severity classes) to identify "at risk" schedules or reliabilities. Appropriate corrective actions are then taken. The comparison is also used to guide release from component test to system test, system test to beta test, or beta test to general availability.

Certification testing uses a reliability demonstration chart [3], illustrated in Figure 4. Failure times are normalized by multiplying by the appropriate failure intensity objective. Each failure is plotted on the chart. Depending on the region in which it falls, you may accept or reject the software being tested or continue testing. Note that Figure 4 shows a test in which the first two failures indicate you should continue testing, and the third failure recommends that you accept the software.

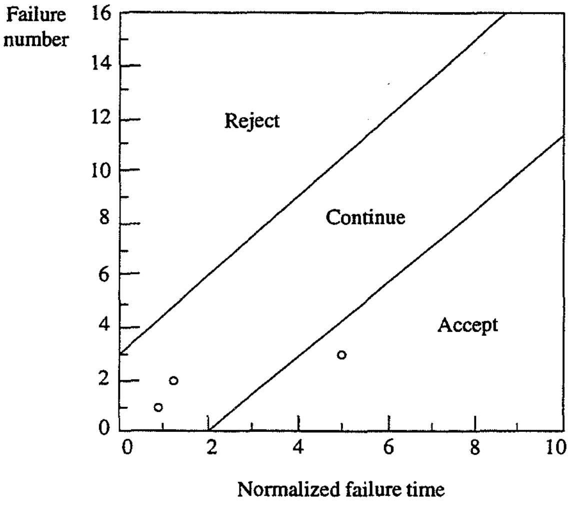


Figure 4. Reliability Demonstration Chart

Rehearse Customer Acceptance Test

In many cases, your customer will require an acceptance test of the system you are delivering. In this case, you may want to conduct a rehearsal, using the certification techniques just described.

Conclusion

Practitioners have generally found software-reliability-engineered testing unique in providing a standard proven way to engineer and manage testing so you can be confident in the reliability of the software-based system you deliver as you deliver it in minimum time with maximum efficiency.

References

[1] Musa, J. D., Iannino, A., and Okumoto, K., *Software Reliability: Measurement, Prediction, Application*, McGraw Hill, New York, 1987.

[2] Lyu, Michael (Editor), *Handbook of Software Reliability Engineering*, Mc-Graw Hill and IEEE Computer Society Press, scheduled for publication December 1995..

[3] Musa, J. D., op. cit., pp. 201-203.

Software-Reliability-Engineered Testing (SRET)

John D. Musa
AT&T Bell Laboratories
Murray Hill, NJ
j.d.musa@att.com

Software-Reliability-Engineered Testing (SRET)

1. What is SRET?
   - Testing that is designed and guided by reliability objectives
2. Why SRET?
   - SRET is unique in helping testers insure necessary reliability in minimum delivery time and cost
   - SRET increases tester efficiency and reduces risk of angry customers
   - Standard, proven practice

SEW-2

Software-Reliability-Engineered Testing (SRET)

3. Where SRET?
   - All software-based systems and their major components
   - Component libraries
4. When SRET?
   - Entire system life cycle, all releases, with focus on system test
5. How SRET?
   - This talk provides overview

SEW-3

SRET – A Valuable Standard, Proven Practice Based on Software Reliability Engineering

SRE and hence SRET is:

1. AT&T Best Current Practice since 5/91 (based on widespread practice, documented strong benefit/cost ratio, probing review)
2. Part of standard software development process (undergoing ISO certification) since 4/92 in Operations Technology Center of Network Services Division
3. McGraw-Hill handbook to be published 1995

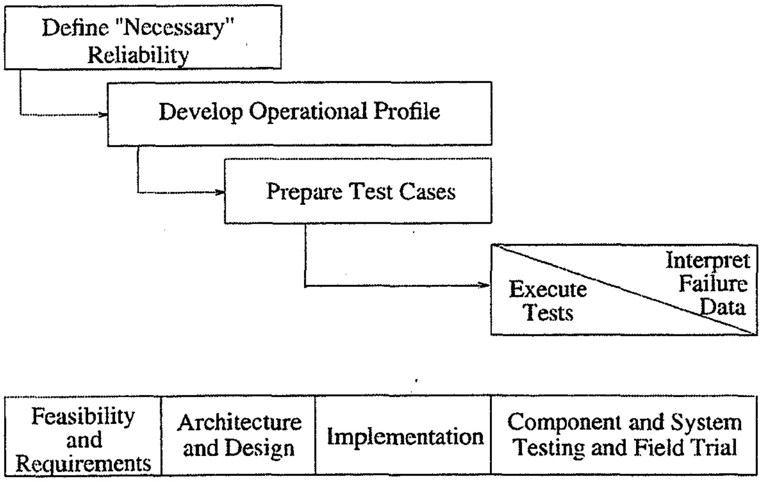
SEW-4

SRET – A Valuable Standard, Proven Practice Based on Software Reliability Engineering

4. AIAA standard published 1993
5. IEEE standards in process
6. AT&T Bell Laboratories President's Quality Award: 4 of 5 software winners used SRE
7. Malcolm Baldrige National Quality Award – 1994 won by AT&T business unit with most SRE use (Consumer Communications Services - Network Services Division)

SEW-5

SRET Process


Define "Necessary" Reliability
Develop Operational Profile
Prepare Test Cases
Execute Tests
Interpret Failure Data
Feasibility and Requirements
Architecture and Design
Implementation
Component and System Testing and Field Trial

SEW-6

Define Necessary Reliability

1. Decide which systems need reliability verification besides product
   - A. Components of unknown reliability
   - B. Supersystems with high interactivity
2. What type of testing does each system need (can be both)?
   - A. Development testing: tries to remove faults; tracks failure intensity, taking corrective action and guiding release, typically used for software developed by your organization
   - B. Certification testing: no debugging, measures reliability to accept or reject software

SEW-7

Define Necessary Reliability

3. Determine which system modes (distinct patterns of system usage) need reliability verification
4. Define failure with severity classes
5. Set failure intensity objectives (per severity class)

SEW-8

Define Failure with Severity Classes

1. failure: departure of program operation from user requirements

2. fault: defect in program that causes a failure when executed

3. severity class: impact of a failure on human life, cost, or service

SEW-9

Set Failure Intensity Objectives (Per Severity Class)

1. failure intensity: failures per unit execution time

2. execution time: actual time used by processor

3. Set failure intensity objectives based on analysis of specific user needs, existing system reliability and user satisfaction, competitor capabilities

SEW-10

Develop Operational Profile

1. operation: externally and independently initiated complete task performed by a system

   Illustrations: command, transaction, processing of external event, administrative housekeeping task

2. operational profile: set of operations and associated probabilities of occurrence

Illustration: Airline reservation system	
OPERATION	PROBABILITY
Reservation: single leg flight	0.6
Reservation: flight with single connection	0.3

SEW-11

Prepare Test Cases

For each system mode:

1. Select test cases
   - A. Operation
   - B. Run

     ILLUSTRATION: Reservation for specific single leg flight, class, person
2. Write test case scripts
3. Provide for recording operations executed

SEW-12

Execute Tests

1. Test each system mode separately
2. Identify failures
3. Record execution times and failure severity classes

SEW-13

Interpret Failure Data – Development Testing

For system and each system mode:

1. Estimate total and per severity class failure intensities periodically by executing a reliability estimation program
   - A. Input failure times or failures in period since last failure
   - B. Read present failure intensity output
   - C. Plot present failure intensity vs calendar time
2. Analyze trend behavior of failure intensities

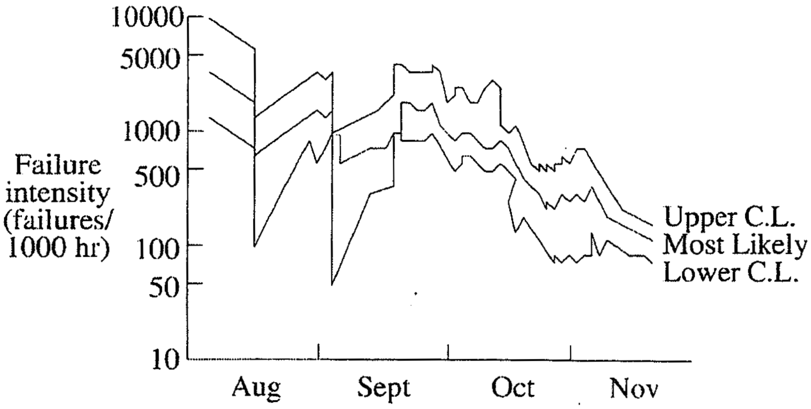
SEW-14

Read Present Failure Intensity Output

	75% CONF. LIMIT - LOWER	MOST LIKELY	75% CONF. LIMIT - UPPER
PRESENT FAIL. INT.	542.7	743.5	1008

SEW-15

Analyze Failure Intensity Trend Behavior



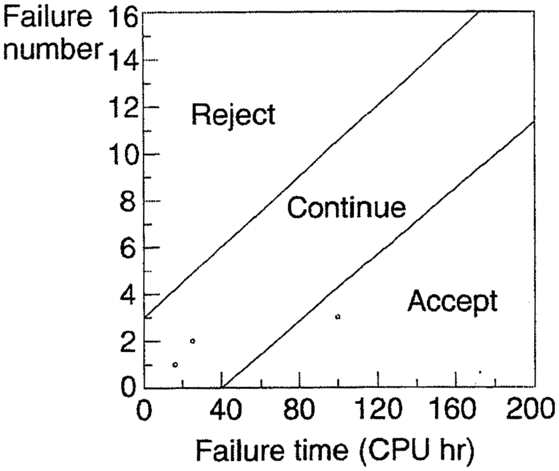
SEW-16

Interpret Failure Data – Certification Testing

After each failure, generate reliability demonstration chart for each severity class of each system mode

SEW-17

Reliability Demonstration Chart



FAIL. NO.	FAIL. TIME	DECISION
1	15	Continue
2	25	Continue
3	100	Accept

SEW-18

Conclusions

1. SRET is unique in providing you with a standard proven way to engineer and manage your testing so you can be confident in the reliability of the software-based system you deliver as you deliver it in minimum time with maximum efficiency.
2. SRET is a vital skill for being competitive

SEW-19

To Explore Further

1. Musa, Iannino, Okumoto; *Software Reliability: Measurement, Prediction, Application,* McGraw-Hill, 1987.
2. Musa, J. D., "Software Reliability Engineering," *Duke Distinguished Lecture Series Video,* University Video Communications, 415-813-0506
3. Musa, J. D., "Operational Profiles in Software Reliability Engineering", *IEEE Software,* March, 1993.
4. Lyu, M. (Editor), *Software Reliability Engineering Handbook,* McGraw-Hill, to be published 1995.

SEW-20

AT&T Bell Laboratories

Reusing Software Reliability Engineering Analysis from Legacy to Emerging Client/Server Systems

by

James Cusick
AT&T Bell Laboratories
Middletown, NJ

and

Max Fine
AT&T Business Communication Services
Somerset, NJ

November 29, 1995

Reusing Software Reliability Engineering Analysis from Legacy to Emerging Client/Server Systems

1. OVERVIEW

Using SRE (Software Reliability Engineering), AT&T TeleConference Services Development controlled reliability rates, decreased testing costs by 89%, and realized a 40% increase in development productivity, over a three year period. These improvements resulted from simultaneously improving development processes and deploying new tools and technologies while reengineering a legacy distributed platform to a new client/server system that employed Object-Oriented technology. Software metrics were applied to measure and track reliability and productivity. This paper presents the techniques used to accomplish these results and expands on earlier findings (Cusick, 1993).

2. CHANGES AND RESULTS SUMMARIZED

During 1992 we experienced a "quality under-run" during deployment of a distributed system used in support of audio teleconferencing. In reaction we investigated current development practices. Reliability struck us as the most important customer-visible characteristic of our product. SRE provided a means to quantify our product reliability and monitor changes over time including over system releases.

Like many multi-project study reports (Kitchenham, et. al., 1995) the impact of SRE is hard to accurately assess given that several process and technology changes were also introduced. Nevertheless, we found SRE provided a constant means of monitoring our development effectiveness quantifiably. ***Table 1*** presents three distinct stages of process evolution the team experienced, the changes implemented in each stage, and the results measured.

3. THE BASELINE

Prior to 1990, the development organization had produced several generations of successful systems. Staff turnover lowered average experience levels leading up to 1992. The development environment was characterized by older technologies and poorly defined processes. The result in 1992 was deployment of a new system with a high occurrence of customer perceived software failures.

4. THE LEGACY IMPROVEMENTS

Management reacted to this stage of problematic development practices by increasing the staff levels of dedicated test personnel and exploring better development methods. Detailed test phases and associated entrance and exit criteria were reestablished.

4.1 Initial SRE Deployment

A study on applying software metrics recommended deploying SRE based on a key attribute of concern to AT&T, namely reliability. The concept of proactively monitoring software failure rates (which at the time were being reported by irate customers after-the-fact) was more attractive than measuring productivity (which customers perceive indirectly through the cost of a conference call).

	STAFF	***PROCESS***	***TECHNOLOGY***	***RESULTS***
BASELINE 1990-1992	Inexperienced Management and Architect	Ill-Defined Process Stages	Homegrown Emphasis	Uneven Quality
		One-pass Integration Test at end of development	Archaic Tools	High Failure Rates
LEGACY IMPROVEMENTS 1992-1993	Increased Test Staff	Well Defined Phase Criteria	Automated Test Case Management	Controlled Reliability Level
	Build Management Support	SRE Introduced with other metrics	SRE Support Tools with training	89% Cost Saving in Field Trial Phase
CLIENT/SERVER CHANGES 1993-1995	Improved Staff (doubled average experience level)	Incremental Development with Early-Often Integration driven by Operational Profile	Object Technology introduced with tools and training	Integration test results indicate probable increase in reliability
		On-Line Up-to-date Documentation	High Reliability Components (Reused and Commercial)	Improved Productivity (+40%)
		Expanded Design Reviews & Walkthroughs	ROS (Reduced Operation Software)	No impact on cycle times (i.e., same as prior systems)

Table 1: Changes and Results Summarized

SRE calls for the development of an Operational Profile, up-front failure classification, calibration of an execution time metric, and the collection and analysis of failure occurrences (Musa, 1987). These steps result in a reliability metric which can be expressed as:

$$R(\tau) = exp(-\lambda\tau)$$

where

$R(\tau)$ = *reliability for time* τ
exp = e^x
λ = *failure intensity*
τ = *execution time*

In order to derive a reliability measure for a system each of these variables must be supplied. We now discuss how we generated these data elements and how we governed our development process using this standard reliability metric and an Operational Profile.

4.2 Execution Time Metric

An attempt was initially made to use CPU usage for the execution time metric. Collection of these data would have required supporting software to be developed. Further, the meaning of reliability statistics based on CPU usage was not intuitive to the team. Instead, we used an approximation of execution time, the number of conferences processed each day by our system (Ackerman, 1993). The use of *conferences-run* as the execution time metric was the most important choice in our SRE process.

Our systems have a built in pulse. Failures per conferences run is a concept we could grasp immediately and derive easily. During laboratory tests simple database queries collect this data point. Production reports already conveniently grouped both *conferences-run* and the next required metric; observed failures.

4.3 Failure Tracking

Our earliest efforts to track failures counted only failures reported by the application programs. While these failures are important the team decided that they are not as important as the failures observed by users. Furthermore, accurate counting would require software development resources beyond our budget. Production environment failures had always been carefully tracked and reported by the support team. These failure reports now arrive weekly via electronic mail to the development team for analysis. This required virtually no additional cost to implement.

Significantly, we skipped the up-front step recommended by standard SRE practice of conducting a thorough severity classification of failures. Instead we define failures as any behavior of the software which deviates from its planned or expected behavior. The working assumption in collecting failure reports from the field is that the users best determine what is and what is not a meaningful failure. We did not find this approach to negatively impact the usefulness of the SRE techniques.[1]

4.3.1 Code Counting, Defect Prediction, and Function Point Backfiring

In parallel with the deployment of SRE several other metrics were introduced. In order to estimate potential defects we conducted Lines of Code (LOC) analysis (Musa, 1987). This also served the purpose of feeding a Function Point "Backfire" which estimates Function Points from LOC and other data (Jones, 1991, 1992). As it turned out the fault prediction formulas provided by Musa gave a closer approximation of defects than those of Jones. However, Backfiring proved valuable in estimating productivity. These up-front defect predictions complemented the reliability calculations in gauging the readiness of the software for release when compared to our observed defects.

4.4 Defining the Operational Profile

Using an Operational Profile to guide system testing generates software usage in accordance with the probabilities that similar functional usage patterns will be followed in production. Basing your testing on an operational profile assures that the most heavily used functions have been adequately tested (Musa, 1993).

Data collected from our legacy production system provided historical usage patterns for nearly all system functions. The granularity of the functional breakdown provided adequate direction to build test runs in accordance with system modes and user usage patterns (***Table 2 & 3***). Probabilities of a particular function being used is calculated in relationship to conferences run. For example, if 10 conferences are run per hour, a relationship to new reservations created in that hour can be determined (e.g., at least 10 conferences must have been reserved). Testing based on these observations was conducted during the system test phase.

SYSTEM MODES	**Utilization**
1: Busy Full	0.35
2: Busy Degraded	0.01
3: Slack Full	0.63
4: Slack Degraded	0.01
TOTAL	**1.00**

Table 2*: Legacy Operational Profile, System Modes*

USERS TYPES	**Utilization**
1: Participants	0.70
2: Hosts	0.08
3: Attendants	0.20
4: Administrators	0.02
TOTAL	**1.00**

Table 3*: Legacy Operational Profile, Utilization by User Types*

[1] We did eventually classify failures as customer affecting or non-customer affecting.

Significantly, the breakdown of usage across system modes, users, and functions, resulted in a dramatic shift of test emphasis. In the past we had neglected off-peak scenario testing. However, the system modes indicated that the system was normally in a slack state. As a result of testing in slack mode commensurate with our usage profile we found additional defects which only appeared when the system was lightly loaded. Also, the analysis indicated that for this particular system our end-customers (participants and hosts) generated far more use of the system than did our attendants (operators). This ran contrary to the team's past test focus.

4.5 Tools

Our Legacy environment refit included the development of a semi-automated test case management tool. We also deployed a PC equipped with the AT&T SRE TOOLKIT© for data storage and to conduct SRE calculations. Failure and execution time data are converted and typed into the PC. Using simple UNIX™ shell scripts accomplished much of the code counting for defect predictions and Function Point Backfiring. Thus the investment in tools in the Legacy stage was minimal.

4.6 Legacy Improvement Results

Our "quality under-run" could now be quantified. Armed with our new techniques we were able to view the reliability of our production version, maintain or better that level in the test lab, and observe the production reliability of the new release. ***Figure 1*** below charts the failure intensity of several production releases of the legacy system during 1992 and 1993. Several major releases have followed, each resulting in incremental increases in reliability.

Note the fact that our field trial tests incur high labor costs. SRE allowed us to use a shortened beta cycle thereby yielding an 89% monetary cost savings in testing. Furthermore, we decreased overall failure rates by 54% while simultaneously increasing customer traffic load on the system 34% during 1993.

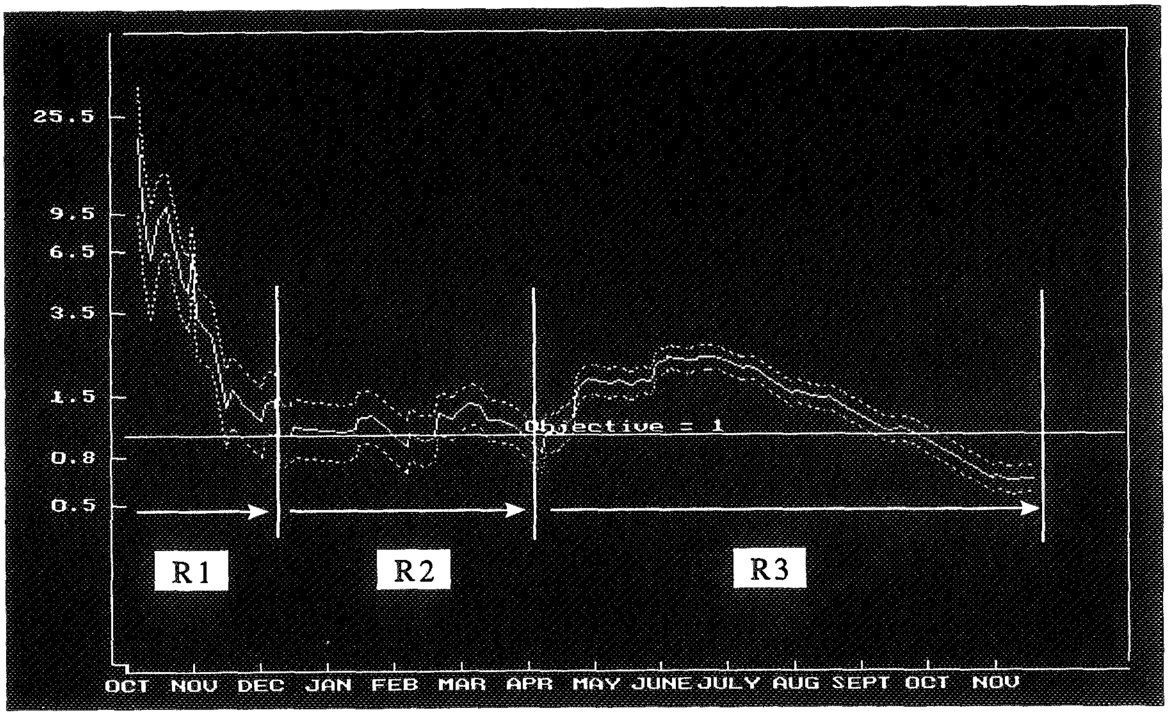


Figure 1: Failure Intensity over Multiple Production Releases: *The Y-axis shows failure intensity on an exponential scale. The X-axis shows calendar time. The dashed lines indicate a confidence factor. The earliest release reflects pre-SRE versions. Variance in the latest release results from differences between test lab predictions and actual production environment conditions.*

The costs of deploying SRE within this Legacy system environment included:

- 1.5 month staff effort for start-up including: process research; 3 weeks training; data collection & analysis.
- 1 staff day per week ongoing effort.

5. THE C/S CHANGES

Concurrent with these legacy changes, business needs required us to abandon our newly improved system and deploy yet another distributed conferencing system. This time we would build the system with the latest Client/Server (C/S) architectures. We would also be in a position to leverage our system development improvements, reliability metrics, and new Object-Oriented methods and techniques.

5.1 Old Dogs, New Tricks

Staff experience plays a key role in successful software development (Arthur, 1985; Pressman, 1992). As ***Table 4*** indicates, the new C/S team possessed roughly double the years of experience of the Legacy team and had on average built at least one such distributed conference system. Thus, we can only agree with the many previous findings on the link of staff experience to successful software development. Nevertheless, we would like to propose that teaching old dogs new tricks may result in higher reliability than if they had used the same old techniques.

5.2 New Tricks Identified

Infrastructure changes were required in order to develop this new C/S system. New workstations, PCs, servers, and Operating Systems were deployed. Object-Oriented methods were reviewed and OMT (Rumbaugh, 1991) was selected.[2] CASE tools, development environments, and new languages were chosen. An on-line and up-to-date documentation repository was established and administered. Detailed project plans including metrics-based estimates were developed. These plans included extensions to our existing design review process. All this work set the stage for several significant extensions to our previous work with SRE.

TEAM	Years of Development Experience (Average)	Number of Similar Systems Built (Average)
Legacy	7.6	0.7
C/S	13.1	1.2

Table 4: *Staff experience differences from Legacy to C/S*

5.3 The Operational Profile as Development Guide

With new feature requirements in hand and a vision of the architecture drawn, one sub-team set out to forecast future usage trends. This Operational Profile would be built from the existing one but would need some expansion. Our capacity planning engineer worked closely with us to construct over-all demand predictions for various services needed by the new system.

The significant finding in this re-analysis was a swing away from end-user interactions back to attendant interactions. This was a key finding dictating where we should shift our development and testing efforts (see ***Table 5)***. We also used our new functional profile to guide the prioritization of feature development.

USER TYPE	LEGACY	C/S
1. Attendant	0.20	0.60
2. Participants	0.70	0.30
3. Hosts	0.08	0.06
4. Administrator	0.02	0.02
5. Rejected Callers	Not Previously Understood	0.02
TOTAL	1.00	1.00

Table 5: *Operational Profiles Compared*

[2] Going against the grain our goal in using Object Technologies was *not* reuse but instead flexibility of future feature introduction.

5.4 Get It Working and Keep It Working

Knowing which features to build first can act as a beacon when spiraling through development. The concept of incremental development has been well understood and documented starting at least with Brooks (1975) and Boehm (1982). Recently Microsoft reports frenetic rates of integration and automated regression testing (Booch, 1995). We found such incremental development, with early integration testing, can be even more beneficial when coupled with Operational Profile driven testing (also termed "operational development", Musa 1993). Our approach on the C/S version of this system was to develop the overall architecture of the system using Object-Oriented Analysis and Design. We then built up functionality recursively across the entire system using Object-Oriented Programming.

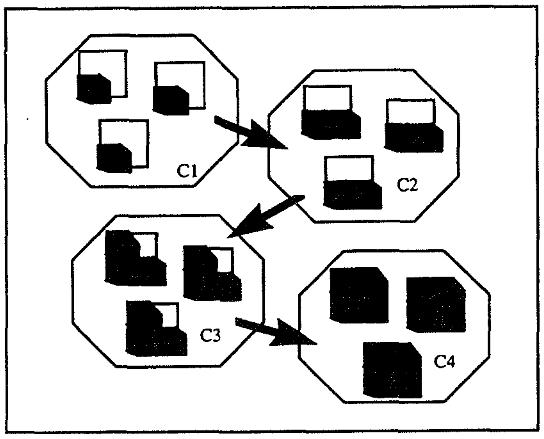


***Figure 2**: Incremental Development Driven by Operational Profile*

Key components are given structural integrity and minimal functionality prior to the initial integration. Load levels selected from the Operational Profile are then used to drive early testing. Successive iterations introduce additional features and functionality in priority as derived from the Functional Profile. As seen in ***Figure 2***, each component's overall scope is understood prior to the initial cycle (C1). As transitions are made to successive cycles (C2,C3,C4), the system soon becomes robust in lab conditions which closely mimic field operations. A side benefit is that the system quickly forms into something deliverable from a product standpoint. With relatively short lead time a working system can be delivered to system test.[3]

5.5 Want Fewer Bugs? Write Less Code!

Throughout the development process one of our lead analysts often chided us that "code we do not write will never break". More formally stated this is the concept of Reduced Operation Software (ROS) (Musa, 1995). Thus, the design bias was towards infusing the objects with nearly automatic behavior based on context and relationships and not by adding procedural machinations. The combination of clear modeling and succinct coding translated into a feature-rich system with fewer crevices in which defects could lodge. Amusingly, the system provided *too much* functionality! In fact one parallel team requested that we turn-off certain features because they had yet to implement the required cooperative process needed for deployment.

Several technology choices supported our mission to develop less code but deliver more functionality. From the beginning we sought out commercially available class libraries. Doing so, in effect, saved months of development effort. In previous efforts we had created custom interface kits to support our computing platform. Moving onto new platforms allowed us to leverage the broad range of tools on the market. This eliminated much development effort (but not all). PC development tools place emphasis on particular computing metaphors (e.g., office document processing). These tools penalize developers who work outside the favored model (in our case telecommunications). Nevertheless, we saved time and gained functionality. Finally, our existing code base provided us with many libraries, functions, code fragments, and utilities for the new system. These components were already proven in production.

[3] Recently, project members reported that the incremental approach taken has been modified. Instead of grouping several features for development in a fixed time interval, schedules now allow feature at a time development across all system components. This has been reported to be much more workable than our earlier attempts at wide scale parallel feature development.

5.6 Preliminary C/S Results

By observing *conferences-run* as the execution time metric and plotting failures observed during integration test runs, our preliminary reliability calculations for the C/S system place it on par with our existing production Legacy system - *prior to system test.* ***Figure 3*** shows failure intensity for the new C/S system during a recent one week integration test phase.

These results must be understood in the context of an emerging system. At the time of this writing the new system was being prepared for a December 1995 release to system test. We believe that results from the more rigorous system test phase and the eventual production environment findings will follow this positive trend.

5.6.1 C/S Productivity Comparison

Previously, our systems required 700 to 1200 Function Points to implement. Productivity figures in Function Points per Staff Month for the entire life-cycle of the Legacy system are presented in ***Table 7*** along with the data on the C/S system though early 1995. The benefits associated with this nearly 40% increase in productivity are partly offset by training costs, ramp up time of 3-6 months for most staff members, and hardware costs. This productivity gain may seem modest but in conjunction with our predicted reliability improvements the total investment has an appreciable return. This leaves out of the equation the business implications of delivering new services.

SYSTEM	Function Points	Productivity (FP/SM)
LEGACY	788	3.10
C/S OO	623	4.34

Table 7: Productivity Comparison

6. LIMITATIONS

We recognize that some limitations and engineering approximations in our SRE program remain. Relying only on the accuracy of the trouble incidence reports results in under-reporting of failures. However, we tend to view this low but consistent method of reporting failures by the users as an adequate barometer of reliability.

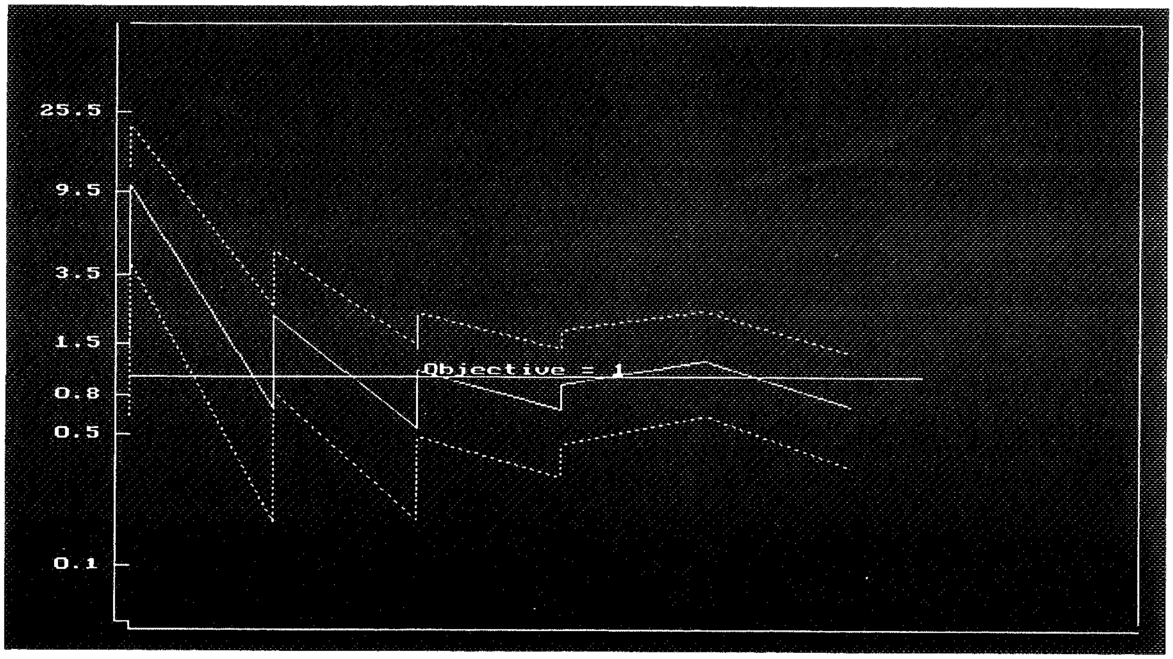


Figure 3: Failure Intensity for Emerging C/S System: *The Y-axis shows failure intensity on an exponential scale. The X-axis shows calendar time during a one week integration test phase. The dashed lines indicate confidence factors.*

Furthermore, we encountered problems extending our analysis to test time prediction. While the projections of the number of defects remaining and reliability levels were useful, the staff time estimates produced by SRE were unusable. The staff time and calendar time estimates put delivery of the system many years in the future. As the lab results neared the failure objective, the time estimates suddenly dropped to zero.

Also, one goal we *failed* to meet was a reduction in cycle time. We intended to deliver the first phase within 18 months of concept formulation. Instead a scaled back initial phase was deployed after 24 months. The follow on phase, including complex service reengineering, proprietary hardware development, and the software development described here, are nearly on schedule but have required some feature trimming to stay on track. Interestingly, these cycle times are consistent with our past delivery intervals.

7. CONCLUSIONS

Neufelder (1993) among others proposed a set of factors influencing software reliability. These factors include methodologies, tools, complexity, testing, languages, schedule, staff experience, and organization. We have found several of these factors to be at work during our system development efforts. SRE provided a set of guiding principles for development prioritization and effort allocation within the two technologically different development environments discussed above. Our gains were not produced by SRE alone. Instead as we made changes in many development activities we were able to measure the impact as a net positive due to our application of SRE and other metrics. Without SRE we would have no yardstick for measuring success or a technique for steadying our transition from one development approach to another.

8. ACKNOWLEDGMENTS

Many AT&T TeleConference Service Development team members assisted our SRE initiative including Dave Cole, Jerry Pascher, Jerry Manese, Joe Chacon, Bob Stokey, Grant Davis, Rao Karanam, Andy Johnson, Bill Masson, Jay Dobin, Jack Samley, and Dottie Bradley. Frank Ackerman now of Octel provided jump-start support. Jim Widmaier of NSA, along with Willa Ehrlich, Ray Sandfoss, and John Musa of AT&T Bell Laboratories provided many useful comments on this paper. Alberto Avritzer of AT&T Bell Laboratories was instrumental in the origination of this document.

9. REFERENCES

[1] Ackerman, F., *"Software Reliability Engineering Applications"*, **AT&T Technical Education**, February, 1993.

[2] Arthur, L., **Measuring Programmer Productivity and Software Quality**, Wiley-Interscience, 1985.

[3] Boehm, B., **Software Engineering Economics**, Prentice-Hall, Engelwood Cliffs, NJ, 1981.

[4] Booch., G., *"The Microsoft Effect"*, Object Magazine, SIGS Publications, Inc., October, 1995.

[5] Brooks, F., **The Mythical Man-Month: Essays on Software Engineering**, Addison-Wesley, Reading, MA, 1975.

[6] Cusick, J., *"Reliability Engineering for System Testing and Production Support"*, **Proceedings of the 4th International Conference on Applications of Software Measurement**, Orlando, FL, November, 1993.

[7] Jones, C., **Applied Software Measurement: Assuring Productivity and Quality**, McGraw Hill, Inc., New York, 1991.

[8] Jones, C., *"Applied Software Measurement"*,**AT&T Technical Education**, November, 1992.

[9] Kitchenham, B., Pickard, L., Pfleeger, S., *"Case Studies for Method and Tool Evaluation"*, **IEEE Software**, July 1995.

[10] Musa, J. D., Iannino, A., and Okumoto, K., **Software Reliability: Measurement, Prediction, Application**, McGraw-Hill, 1987.

[11] Musa, J. D., *"Operational Profiles in Software-Reliability Engineering"*, **IEEE Software**, March 1993.

[12] Musa, J. D., *"The Operational Profile"*, **Proceedings of the NATO Advanced Science Institute**, Antalya, Turkey, June 1995, (to be published by Springer-Verlog, Berlin).

[13] Neufelder, A. M., **Ensuring Software Reliability**, Marcel Dekker, NY, 1993.

[14] Pressman, R., **Software Engineering: A Practitioner's Guide**, 3rd ed., McGraw-Hill, NY, 1992.

[15] Rumbuagh, J., et. al., **Object-Oriented Modeling and Design**, Prentice-Hall, Engelwood Cliffs, NJ, 1991.

A TALE OF TWO SYSTEMS:

eusing Software Reliability Analysis from egacy to Emerging Client/Server Systems

Twentieth Annual NASA SEL Software Engineering Workshop, Greenbelt, MD

November 29-30, 1995

AT&T

James Cusick
AT&T Bell Laboratories
James.Cusick@att.com

ere We Started

- Small teams, date driven, resource limited, uneven quality
- SRE's Promise:
  - Quantitative measure of current quality
  - Control release of future software
  - Demonstrate quality improvement

anges to Legacy Environment

- ACQUIRING THE PROCESS
  - Try, try, and try again (getting started)
  - Training, tools, process definition
  - Conferences-run rate as execution metric
- ANIMATING THE PROCESS
  - Data collection, baselines, objectives
  - Operational Profile driven testing
  - Release and production monitoring

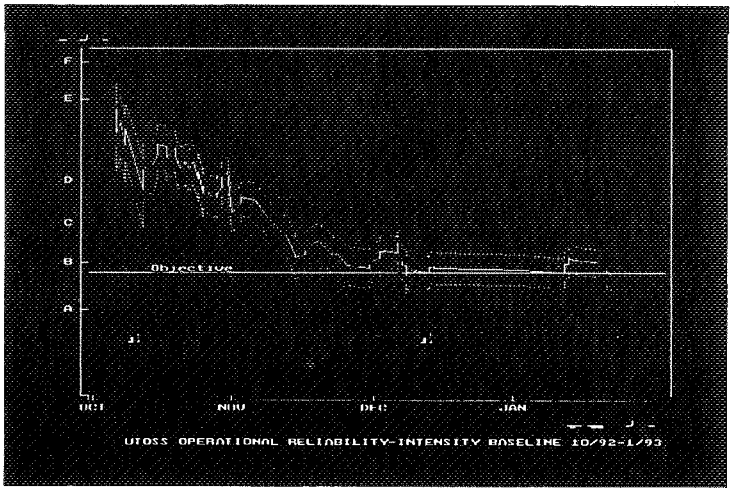
gacy Operational Profile

- USER MODES
  - Busy 36%
  - Slack 64%
- USER TYPES

• Participants	0.70
• Hosts	0.08
• Attendants	0.20
• Admin istrators	0.02

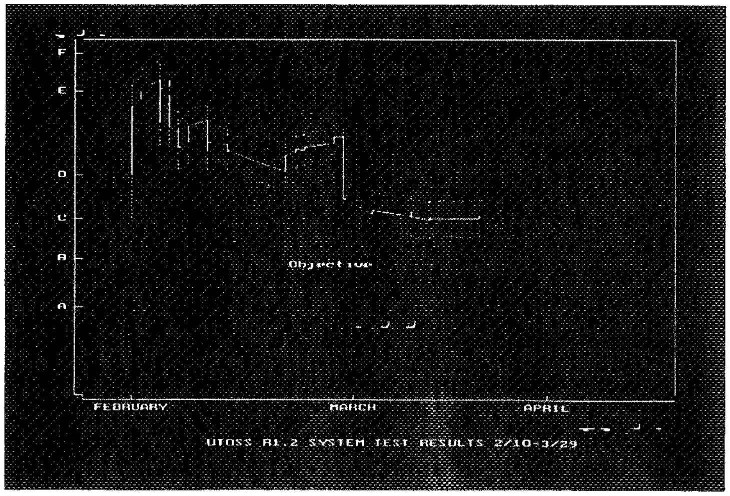
<u>Insight</u>

gacy Production Baseline



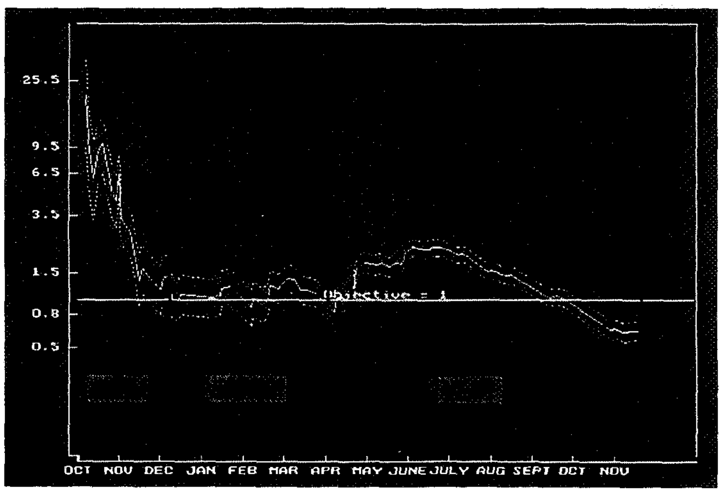
UTOSS OPERATIONAL RELIABILITY-INTENSITY BASELINE 10/92-1/93

stem Test Results



UTOSS R1.2 SYSTEM TEST RESULTS 2/10-3/29

oduction Results



gacy Improvements: The Bill

- COSTS
  - *1.5 Staff months for startup (research, training, analysis, data collection), 1 staff day per week ongoing, $3K training costs*
- BENEFITS
  - *89% Cost Savings in Beta Test*
  - *New version with predictable reliability*
  - *Decreased failure rate 54%; while increasing traffic 34%*
  - *New paradigm for decision making*

anges for C/S Environment

One More Time

- New hardware, new tools
- More experienced, re-trained staff
- Incremental development driven by Operational Profile
- On-line documentation
- Object-Oriented Technologies

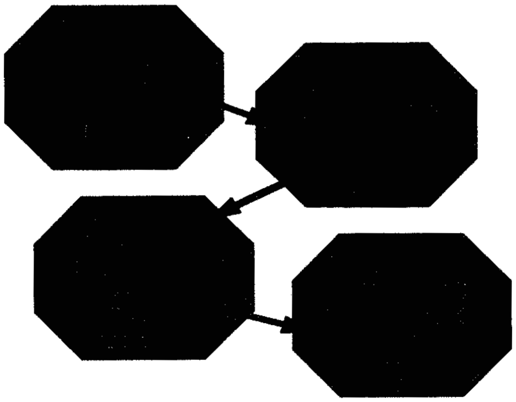
S Operational Profile

USER TYPES	Legacy	C/S
• Participants	0.70	0.30
• Hosts	0.08	0.06
• Attendants	0.20	0.60
• Administrators	0.02	0.02
• Rejected Callers	------	0.02

Insight

cremental Integration

GET IT WORKING AND KEEP IT WORKING

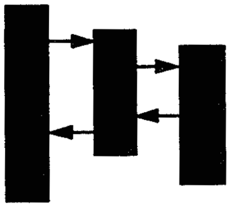


PRIORITIZE WITH OPERATIONAL PROFILE

duced Operation Software

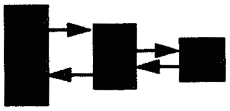
LESS CODE GENERATES FEWER DEFECTS

Data validation at each step



Defects = (LOC)(Defect Density)*
Defects = (100KLOC)(1/K)
Defects = 100

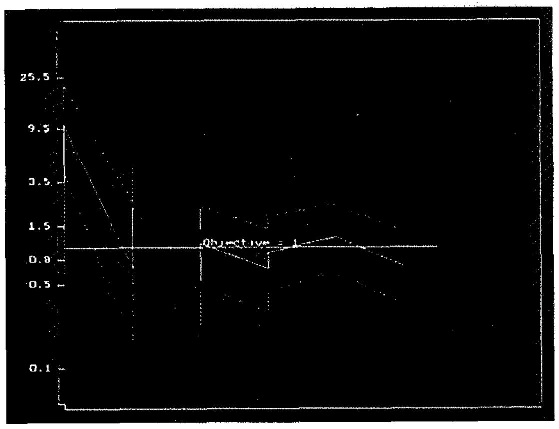
Data validation at needed step



Defects = (LOC)(Defect Density)*
Defects = (10KLOC)(1/K)
Defects = 10

eliminary C/S Results

- egration Test results show same or better reliability compared to Legacy
- Productivity increased 40% in Function Points per staff month
- Zero cost for SRE reuse; hardware and training costs coincidental to redesign



Integration Failure Intensity vs. Calendar Time

nclusions

- This case depends upon SRE <u>and</u> new tools, technologies, and staff
- SRE compatible with Object-Oriented Technologies
- SRE excellent means of determining return on investment in technology
- SRE requires management commitment and local champion to succeed

Special Presentation: Software Engineering Survey

Software Engineering Survey
Jon Valett, NASA/Goddard

Survey Results and Award Presentations

Jon D. Valett
NASA/Goddard Space Flight Center

As part of the 1995 workshop program, significant "behind-the-scenes" contributors were given special award presentations, and the results of a survey were presented. The awards were given to Laura Moleski, John Cook, and Barbara Holmes for their many years of continuous service to the Software Engineering Workshop. All three of these individuals contributed tremendously to the success of many workshops.

To mark the twentieth anniversary of the Software Engineering Workshop, the program committee distributed a questionnaire to everyone on the workshop mailing list (approximately 3500 people with 125 respondents). This questionnaire was similar to one that was distributed at the tenth workshop. The purpose was to obtain information from the respondents concerning many aspects of software engineering. (The questionnaire follows as the first page after this summary). The first slide shows a number of facts about the workshop (many of which were not obtained from the questionnaire).

The second slide compares the attendance profile from 1995 with that of 1996. It also summarizes the answers to many other questions. The attendance profile has not changed drastically, except that the 1996 survey added a function of process improvement. Most of the people in that category probably came from the manager and researcher category of 1985. One other interesting note was the increase in the number of organizations that are collecting metrics data. Finally, in the 1995 survey, a question on whether the organization has a process improvement program in place. 72% of the respondents stated that there was a process improvement program in place in their organization.

The third slide shows people's opinions of how the quality of software has changed within their organizations over the past 5-10 years. The majority of people thought that their quality was improving both in 1995 and in 1985, but people were slightly less optimistic in 1995. Perhaps, quality is improving slightly less noticeably then it was 10 years ago.

The final slide shows the results of the question on the greatest improvement and biggest disappointment in the state of the practice over the past number of years. The answers over the two surveys were not significantly different. In 1995, 43% of the respondents felt that methods, practices, and tools were the greatest improvement in the state-of-the-practice. Perhaps this indicates the major emphasis on process and methods over this ten year period. Certainly, the software engineering community has had a significant focus on process during this time period.

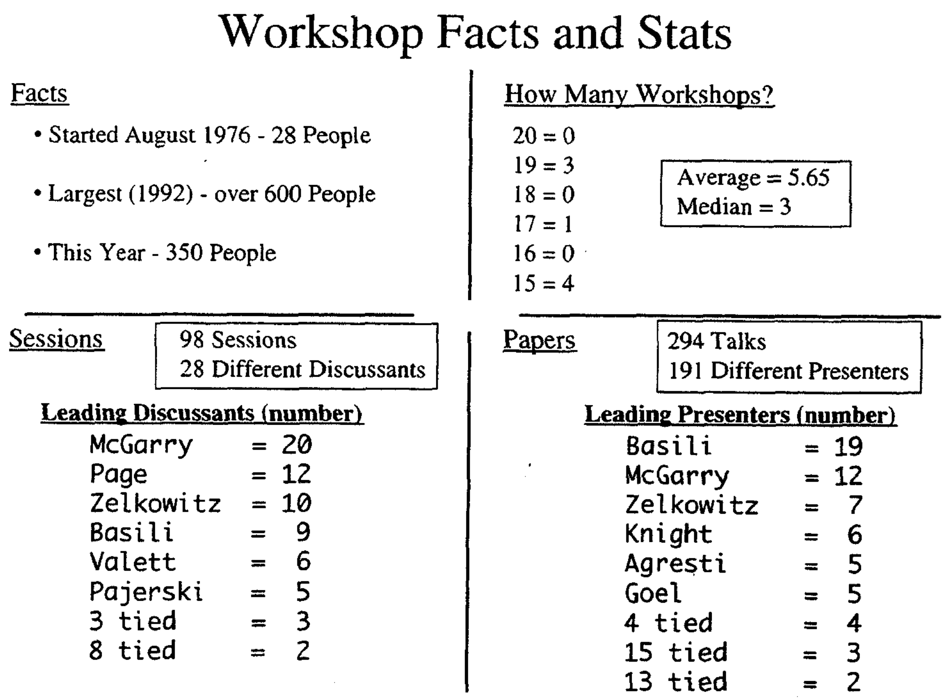
Overall, the survey was quite successful in collecting a variety of opinions about various software engineering issues.

Acknowledgement

I would like to thank John Cook for collecting the data and summarizing the results from the questionnaire.

20th Software Engineering Workshop Survey Results and Award Presentations

Jon D. Valett
NASA/Goddard Space Flight Center



Workshop Facts and Stats

Facts

- Started August 1976 - 28 People
- Largest (1992) - over 600 People
- This Year - 350 People

How Many Workshops?

20 = 0
19 = 3
18 = 0
17 = 1
16 = 0
15 = 4

Average = 5.65
Median = 3

Sessions

98 Sessions
28 Different Discussants

Leading Discussants (number)

McGarry	=	20
Page	=	12
Zelkowitz	=	10
Basili	=	9
Valett	=	6
Pajerski	=	5
3 tied	=	3
8 tied	=	2

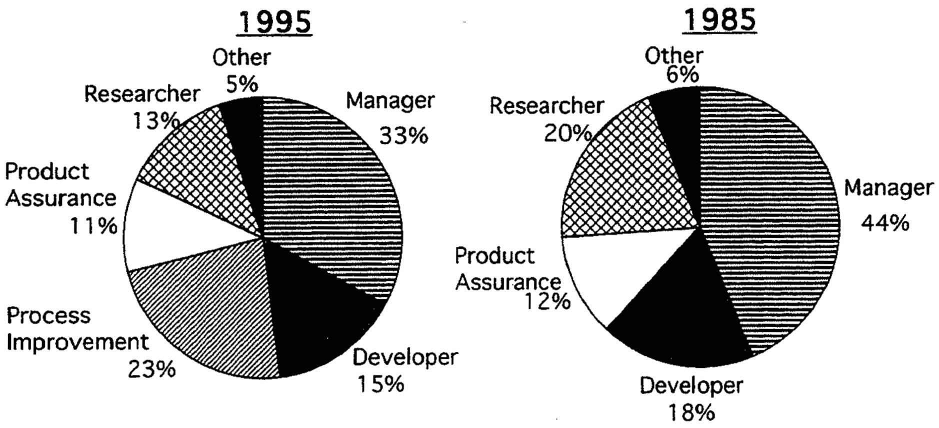
Papers

294 Talks
191 Different Presenters

Leading Presenters (number)

Basili	=	19
McGarry	=	12
Zelkowitz	=	7
Knight	=	6
Agresti	=	5
Goel	=	5
4 tied	=	4
15 tied	=	3
13 tied	=	2

Attendance Profile



Collect Data?

1995 = 85%
1985 = 69%

Use SEL Results?

1995	1985
Yes = 79%	Yes = 68%
No = 6%	No = 8%
N/A = 15%	N/A = 24%

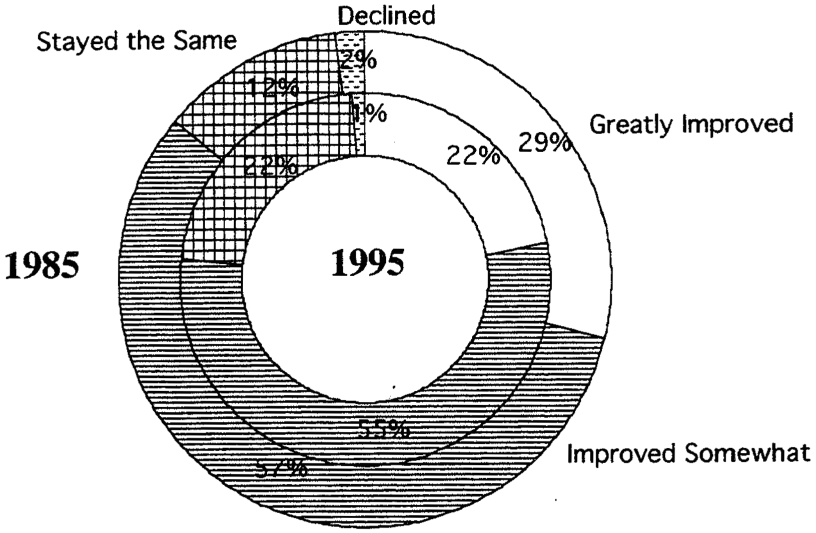
Process Improve. Program?

1995	Type?

Yes = 72%	CMM = 63%
	QIP/SEL = 26%
	Other = 11%

Quality of Software

(Last 5-10 years)



Quality is improving LESS noticeably than 10 years ago

Greatest Improvement/Disappointment

- Improvement in State-of-the-Practice
  - 1995
    - Methods, Practices, Processes = 43%
    - Software Tools = 22%
  - 1985
    - Tools = 31%
    - Methods, Practices = 30%
- Disappointment in State-of-the-Practice
  - 1995
    - Management = 33%
    - Metrics = 23%
    - Tools = 15%
  - 1985
    - Management = 28%
    - Metrics = 24%
    - Methods, Practices = 12%

OMIT
THIS
PAGE

Session 3: Product Evaluation

A Family of User Interface Consistency Checking Tools: Design and Development of SHERLOCK
Ben Shneiderman and Rohit Mahajan, University of Maryland

A COTS Selection Method and Experiences of Its Use
Jyrki Kontio, University of Maryland

Process Enactment within an Environment
Marv Zelkowitz, University of Maryland

November 25, 1995

A Family of User Interface Consistency Checking Tools: Design and Development of SHERLOCK

S8-61

915604

360694

20 P.

Rohit Mahajan and Ben Shneiderman[1]
[1]*Department of Computer Science,*
Human-Computer Interaction Laboratory &
Institute for Systems Research
University of Maryland, College Park, MD 20742 USA
email: mahajan@cs.umd.edu, ben@cs.umd.edu

ABSTRACT

Incorporating evaluation metrics with GUI development tools will help designers create consistent interfaces in the future. Complexity in design of interfaces makes efficient evaluation infeasible by a single consistency checking evaluation tool. Our focus is on developing a family of consistency checking tools to evaluate spatial layout and terminology in user interfaces and make the evaluation process less cumbersome. We have created a dialog box summary table to provide a compact overview of spatial and visual properties of dozens or hundreds of dialog boxes of the interface. Interface concordance tool has been developed to spot variant capitalization and abbreviations in interface terminology. As buttons are most frequent used widgets, a button concordance tool and a button layout table has been constructed. Button concordance identifies variant capitalization, distinct typefaces, distinct background colors and variant sizes in buttons. Button layout table spots any inconsistencies in height, width and relative position between a given group of buttons. A spell checking tools which detects spelling errors in interface terms has also been included in the tool set. Finally, a terminology basket tool has been created to identify unwanted synonyms of computer related terms used in the interface. These tools are integrated together as SHERLOCK, a family of six consistency checking tools to expedite the evaluation process and provide feedback to the designers plus aid Usability Testing.

KEYWORDS: Automated metrics, consistency checking tools, concordance tools, spatial and textual evaluation tools, user interface

1. INTRODUCTION AND PREVIOUS RELATED RESEARCH

Creating user interfaces is a composite procedure involving iterative design, usability testing and evaluation processes (Shneiderman, 1992). Iterative refinement methods like Formative Evaluations can be used to design/redesign the interface from early development stages through completion stage (Hix & Hartson, 1993). Interactive tools like IDEAL (Interface Design Environment Analysis Lattice) support procedures like Formative Evaluations (Ashlund & Hix, 1992). Recent advances in powerful user interface development tools have expedited the interface development process helping both novice and experienced developers. However these expeditiously created designs may be clogged with spatial and textual inconsistencies which cannot be verified by current development tools. These inconsistencies may have a subtle and negative impact on interface usability.

Inconsistencies in spatial and textual style of an interface designed by several designers may result in a chaotic layout. Each designer may have different interpretation of terminology and may use his/her own style of abbreviations and computer terms. Furthermore designers personal preferences on fonts and colors add to the problem in group designs. Such anomalies in terminology and format lead to poor design, ultimately misleading and confusing the user (Chimera & Shneiderman, 1993). Although many organizations are adopting more stringent usability testing standards to monitor quality and layout of the design, better automated evaluation tools are needed which would scan for inconsistencies in the interface layout at early design and development stages, thereby providing an aid to the Usability testing. Also, these automated tools may unearth problems that would be missed by usability testing.

Usability testing is a highly beneficial but costly process when compared with automated evaluation. Prerequisites for these tests may include availability of developed working prototypes, test users and expert evaluators (Sears, 1994). These requirements are hindrances in this very powerful evaluation method. Alternative techniques like Heuristic Evaluations (Nielsen & Molich, 1990) can decrease but not eliminate these requirements. Furthermore usability testing works best for smaller applications. It is practically infeasible to analyze every dialog box in an application with thousands of dialog boxes with the current evaluation methods. Finding anomalies or differences while reviewing thousands of dialog boxes is even hard for expert reviewers who may leave undetected flaws and inconsistencies. In contrast automated evaluation tools can be used in early prototypes (or later iterations) and can detect anomalies across thousands of dialog boxes. These automated tools in addition to detecting anomalies can make interface cleaner and easier to use.

Automated tools for consistency checking are meant to replace the current consistency checking process which is complex, expensive, error prone and manual. These tools can be made independent of platform and development tool, as textual and spatial properties are independent of these constraints. Spatial metrics to check consistencies in alignment, screen symmetry, screen balance, average distance between groups of items, percentage of screen used to display information, average size of groups of items were introduced by Streveler and Wasserman (1987) and were later implemented by Tullis(1988). Furthermore Kim and Foley (1993) used metrics as a constraint for design space and layout style. They developed a tool which generated potential designs for an interface when provided with design specifications and guidelines for metrics. Effectiveness of their metrics has not yet been evaluated.

Evolution of modern user interfaces like multimedia interfaces has sparked research in automated evaluation based on visual techniques. Vanderdonckt and Gillo (1994) proposed five visual techniques: physical, composition, association, ordering and photographic which identified more spatial properties than traditional balance, symmetry, and alignment. These visual properties also include proportion, neutrality, singularity, repartion, grouping, sparing, and simplicity. Dynamic strategies for automated evaluation using these visual techniques have been introduced. (Bodart, Hennebert, Leheureux and Vanderdonckt,1994). Visual metrics introduced above for traditional layout grids and multimedia layout frames have not yet been tested.

Sears (1993, 1994) has developed a first generation tool using automated metrics for both design and evaluation using Layout Appropriateness metrics. The tool AIDE (semi- Automated Interface Design and Evaluator) allows designers to create, evaluate and modify an interface using a single tool. Layout Appropriateness compares layout based on user's task sequences and frequencies. AIDE has demonstrated its effectiveness in analyzing simple interfaces. Currently, studies are being done by Comber and Maltby (1995) in assesing the usefulness of layout complexity metric in evaluating the usability of different screen designs.

2. METRICS EVALUATION USING CANONICAL FORMAT

Our research evolved from the concept of converting interface form files generated by Visual Basic into canonical format files and feeding them as input to the SHERLOCK. The canonical format is an organized set of GUI object descriptions which embrace interface layout and terminology information in a sequence of attribute-value pairs. These canonical formats may be created for other interface development tools like Power Builder, Galaxy, and Visual C++ and by writing a translator program for these tools. SHERLOCK is not specific to Visual Basic and can be used for evaluating interfaces developed by other tools.

2.1 Our Evaluation Method

This research is an extension of previous work (Shneiderman, Chimera, Jog, Stimart and White, 1995) in which we developed spatial and textual evaluation tools. The spatial tool was a dialog box summary table which gave an overview of spatial and visual properties. Each dialog box corresponded to a distinct row and each column a metric. The metrics Aspect Ratio, Widget Totals, Non-Widget Area, Widget Density, Margins, Top-Bottom Balance, Left-Right Balance and Distinct Typefaces formed our metrics column set. This list of metrics was developed by consultation with analysts at University of Maryland and General Electric Information Services to evaluate categories such as spatial layout, alignment, clustering, cluttering, fonts, etc. The textual tool was a concordance built to extract all the words that appear in labels, menus, buttons, etc. in every dialog box. These words were sorted in one file with reference to the dialog boxes containing them. The concordance was to help designers in appropriate word use such as spelling, abbreviation, tense consistency, case consistency, passive/active voice etc.

SHERLOCK "A family of consistency checking tools" was constructed by modifying our previous tools. The metrics of the dialog box summary table have been modified and new metrics have been added after evaluating more interfaces. Further, the tool set has been expanded by adding new tools which in many cases perform exception reporting by outputting the possible anomalies and irregularities in spatial and textual layout. The reports generated by these mini tools require less interpretation, thereby expediting the quick evaluation process and providing feedback to the designer. The designer then must decide whether the spotted inconsistencies are relevant to the particular prototype. We have developed six consistency checking tools:

- dialog box summary table to give an overview of spatial and visual properties of the interface dialogs.
- interface concordance to spot variant capitalization and abbreviation in button widgets.
- button concordance to spot variant capitalization, distinct typefaces, distinct background colors and variant sizes in all the interface buttons.
- button layout table to spot any inconsistencies in height, width and relative position among a given group of buttons.
- interface speller to detect terms used in the interface that are nonexistent in the dictionary.
- terminology basket to provide the interface designer with the feedback on misleading synonym computer terms.

2.1.1 Dialog Box Summary Table

The dialog box summary table is a compact overview of spatial and visual properties of the dozens or hundreds of dialog boxes of the interface. Each row represents a dialog box and each column represents a single metric. Typical use would be to scan down the columns looking for extreme values, spotting inconsistencies, and understanding patterns within the design. The following are the columns of the table:

Aspect Ratio: The ratio of the height of a dialog to its width. Numbers in the range 0.5 thru 0.8 are desirable. Dialogs that perform similar functions should have the same aspect ratio.

Widget Totals: Counts of all the widgets and the top level widgets. Increasing difference between all and top level counts indicates greater nesting of widgets, such as buttons inside containers.

Non-Widget Area: The ratio of the non-widget area to the total area of the dialog, expressed as a percentage. Numbers closer to 100 indicate high utilization, and low numbers (< 30) indicate possibilities of redesign.

Widget Density: The number of top-level widgets divided by the total area of the dialog (multiplied by 100,000 tó normalize it). High numbers greater than 100 indicate that a comparatively large number of widgets are present in a small area. This number is a measure of the 'crowding' of widgets in the dialog.

Margins: The number of pixels between the dialog box border and the closest widget. The left, right, top and bottom margins should all be approximately equal to each other in a dialog, and should also be the same across different dialogs.

Gridedness: Gridedness is a measure of alignment of widgets. High values of x-gridedness and y girdedness indicate the possibility of misaligned widgets. X-gridedness counts the number of stacks of widgets with the same x coordinates (excluding labels). Similarly Y-gridedness counts the number of stacks of the widgets with the same y coordinates. An extension of Gridedness is Button Gridedness where the above metrics are applied to button widgets.

Area Balances: A measure of how evenly widgets are spread out over the dialog box. There are two measures: a horizontal balance, which is the ratio of the total widget area in the left half of the dialog to the total widget area in the right half of the dialog; and the vertical balance, which uses top area divided by bottom area. High value of balances between 4.0 and 10.0 indicate screens are not well balanced.

Distinct Typefaces: Typeface consists of a font, font size, bold and italics information. Each distinct typeface in all the dialog boxes is randomly assigned an integer to facilitate quick interpretation. For each dialog box all the integers representing the distinct typefaces are listed so that the typeface inconsistencies can be easily spotted locally within each dialog box and globally among all the dialog boxes. The idea is that a small number of typefaces should be used for all the dialog boxes.

Distinct Background Colors: All the distinct background colors(RGB values) in a dialog box are displayed. The purpose of this metric is to check if all the dialog boxes have consistent background colors. Multiple background colors in a dialog box may indicate inconsistency.

Distinct Foreground Colors: Similar to distinct background colors, displays all the distinct foreground colors(RGB values) in a dialog box. The purpose of this metric is to check if all the dialog boxes have consistent foreground colors.

This tool was tested with all the four test applications and inconsistencies were revealed in all the applications. A portion of the table from the 51 dialog box University of Maryland, AT&T teaching theater interface (Table 1) is shown below which uses 12 distinct typefaces, 7 background colors and 9 foreground colors. Our programs allowed the designers to check the dialog boxes for inconsistencies and the output of the dialog box summary table revealed anomalies which otherwise may not have been detected. For example the margins were irregular and aspect ratio was variant even for similar styled dialog boxes. The non-widget area varied from single digits to nearly 100%.Thus, some screens were crowded with widgets and others were almost blank showing inefficient screen designs of the application.

No.	Dialog Name	Aspect Ratio (H/W)	-WIDGET-- TOTALS		Non- Widget Area (%)	Widget Density widget/ area	-----M A R G I N S----- (pixels)				--GRIDEDNESS--				--Balances-- Area Ratios		Distinct Typefaces	Distinct Bgrnd Colors
			All	Top-Level			Left	Right	Top	Bottom	Top Level X	Top Level Y	Buttons X	Buttons Y	Horiz (L/R)	Vert (T/B)		
42	review5.frm	1.60	15	7	43.5	64	0	22	0	17	2	4	1	2	2.1	0.4	4	2 5 6
43	single1.frm	1.30	20	9	57.3	58	0	37	0	87	4	5	0	0	2.4	0.6	4	2 5 6
44	time_up.frm	0.67	5	3	79.4	140	0	78	0	31	1	2	1	1	8.4	3.9	4 9	2 5 6
45	time_up2.frm	0.67	4	2	79.4	105	0	78	0	31	2	2	1	1	4.6	2.1	4 9	2 5 6
46	topic2.frm.	0.76	13	8	55.7	55	24	25	8	16	4	5	0	1	0.9	1.2	4	2 5
47	vdmdi3.frm	0.78	72	9	94.2	25	0	0	0	0	1	1	0	1	0.0	0.0	4	2 5
48	winchat8.frm	0.77	19	9	16.5	39	0	0	0	0	3	4	0	0	0.8	0.4	4 11	1 2 5 6
49	winstat.frm	0.92	12	11	89.2	29	0	70	0	24	0	3	0	1	1.6	0.3	4 12	2 5 6
50	zoom.frm	0.47	5	4	14.4	108	3	5	2	17	1	2	0	1	1.1	1.0	4	1 2 5
	Maximum	1.60	72	15	100.0	140	48	381	27	276	5	6	1	3	10.0	10.0		
	Minimum	0.13	1	1	5.0	0	0	0	0	0	0	0	0	0	0.0	0.0		
	Average	0.77	11	5	53.8	48	9	53	6	29	1	2	0	0	2.2	1.7		

DISTINCT TYPEFACES:

1 = Arial 13.5 Bold	2 = Symbol 9.75 Bold	3 = Arial 8.25 Bold	4 = MS Sans Serif 8.25 Bold
5 = System 9.75 Bold	6 = Arial 15.75 Bold	7 = MS Sans Serif 9.75 Bold	8 = MS Sans Serif 16.5 Bold
9 = MS Sans Serif 12 Bold	10 = MS Serif 30 Bold	11 = Times New Roman 12 Bold	12 = MS Serif 12 Bold

DISTINCT BACKGROUND COLORS:
1 = ffffff   2 = ffffffff80000005   5 = c0c0c0   6 = ff   9 = e0ffff   10 = 404040   12 = ffffffff8000000f

DISTINCT FOREGROUND COLORS:
2 = ffffffff80000005   3 = ffffffff80000008   4 =0   6 = ff   7 = ff0000   8 = c000c0   11 = 808080   13 = c000   14 = c00000

Table 1

2.1.2 Interface Concordance

The interface concordance tool checks for variant capitalization for all the terms that appear in buttons, labels , menus, etc. in every dialog box of the interface. This tool outputs strings which have variant capitalization, listing all the variant formats of the string and its dialog box sources. These variant forms are spelling differences and may be acceptable, but they may be something that should be reconsidered. For example the words "MESSAGES" , "messages" ,"Messages" and “msgs” are Variant Capitalization forms of the same word.

2.1.3 Button Concordance

As buttons are one of the most frequently used widgets performing vital functions like “Save”, “Open”, “Delete”, “Exit” etc., checking consistency in their size, placement, typefaces, colors and case usage becomes more important. This tool outputs all the buttons used in the interface, listing the dialog boxes containing the buttons plus fonts, colors and button sizes. The button concordance identifies variant capitalization, distinct typefaces, distinct foreground colors and variant sizes in buttons.

BUTTON LABEL	FORM CONTAINING THE BUTTON	BUTTON TYPEFACE	BUTTON FG_COLOR	BUTTON (H , W)
OK				
	find.frm.cft	1	1	24,112
	grid3.frm.cft	1	1	24,68
	help.frm.cft	1	2	32,72
	help5.frm.cft	1	2	33,73
	list2.frm.cft	1	1	29,92
	listque2.frm.cft	1	1	32,64
	opensav2.frm.cft	1	1	25,73
	time_up.frm.cft	1	1	33,57
Ok				
	form3.frm.cft	1	1	33,105

Table 2

A small portion of the button concordance table from one of the test interfaces is shown above (Table 2). The designer have used both OK and Ok buttons with the height of buttons varying from 24 to 33 pixels and the width varying from 57 to 112 pixels which is an inconsistency. Also, two different foreground colors have been used in the interface for button labels. Fig. 1-3 show some of the dialogs from this interface.

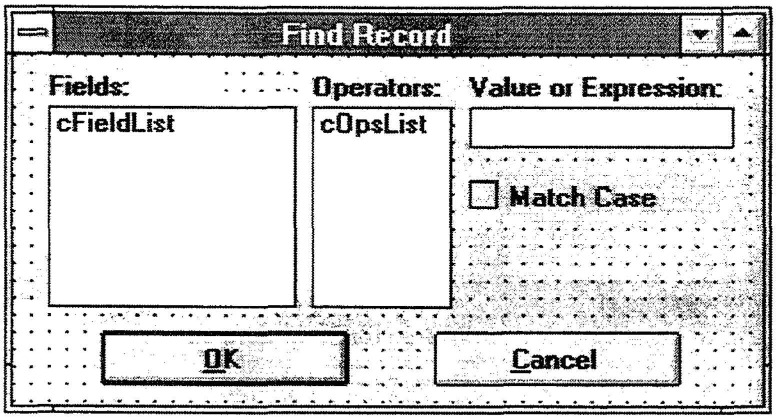


Fig. 1 find.frm.cft

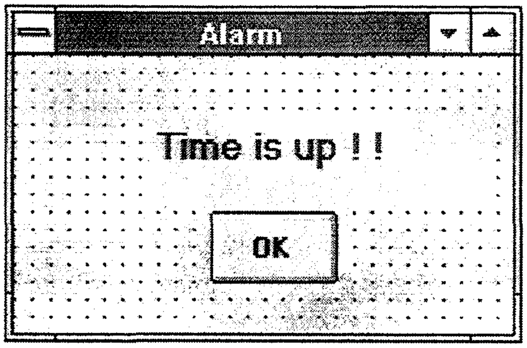


Fig. 2 timeup2.frm.cftt

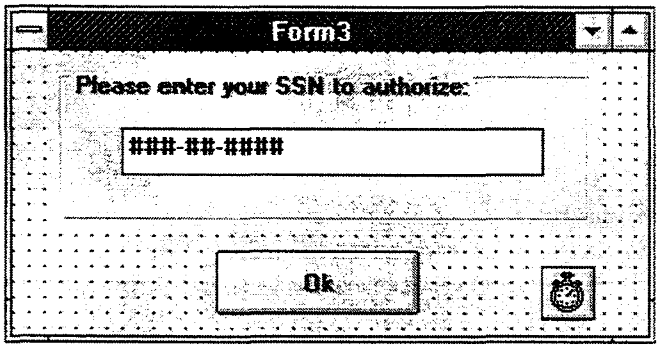


Fig. 3 form3.frm.cft

2.1.4 Button Layout Table

Given a set of buttons that frequently occur together (e.g. OK Cancel, Close, Help), if the first button in the set is detected in the dialog box then the program outputs the height, width and position relative to the first button of every button detected in the set. The relative position of every button detected in the set is outputted as (x ± offset, y± offset) to the first button, where offset is in pixels. Buttons stacked in rows would yield (x± offset, y) relative position and those stacked in columns would yield (x, y± offset). The Button Layout table identifies inconsistencies in button placement, inconsistencies in button terminology plus variant button sizes locally within a dialog box and globally across all the dialog boxes.

Our program reads an ASCII file containing different sets of buttons. These button sets were constructed after analyzing many previously developed interfaces. Variations in terminology were considered while constructing these button sets. Button set (Start Stop Exit) is incomplete as designers may use "Close" , "Done" or "Cancel" instead of "Exit". The set (Start Stop Halt Pause Cancel Close Done End Exit Quit) forms a much better button detector set.. Some of the sample button sets are:

- `OK      Cancel    Close    Exit    Quit    Help`
- `Start   Stop      Halt     Pause   Cancel  Close   Done    End`
`Exit    Quit`
- `Add     Remove    Delete   Copy    Clear   Cancel  Close   Exit`

- Help   Close   Cancel   Exit

A portion of the output using the button set (OK Cancel Close Exit Quit Help) tested with the small 30 dialog box GE application is shown below (Table 3). Inconsistency in height and relative button positions within a button set can be checked by moving across the rows of the table. Inconsistency in height and relative position for a particular button can be spotted by moving down in columns. For example, the height of the "OK" button varies from 22 pixels to 26 pixels and the width varies from 62 pixels to 82 pixels. Also, the relative position between "OK" and "Cancel" buttons varies in all the three forms in which they occur together. In the forms "nbatch.cft" and "systinp.cft" the "Cancel" button is 20 pixels and 13 pixels down respectively from the "OK" button, but in the form "admprof.cft" the buttons occur next to each other in the same row. Also, both the buttons "Cancel" and "Exit"(Fig. 4 and Fig. 5) have been used with the button "OK" essentially to perform the same task which is a terminology inconsistency.

Form Name	**OK**	**Cancel**		**Exit**		**Help**	
	(H,W)	(H,W)	Rel. Pos.	(H,W)	Rel. Pos	(H,W)	Rel. Pos.
admprof.cft	22,68	22,68	x+16, y	22,68	x+98, y		
checkpsw.cft	25,82			25,82	x+18, y	25,82	x+116, y+1
nbatch.cft	25,62	25,62	x-1, y+20	25,62	x, y+66		
systinp.cft	26,72	26,72	x+1, y+13			25,73	x+2, y+48

Table 3

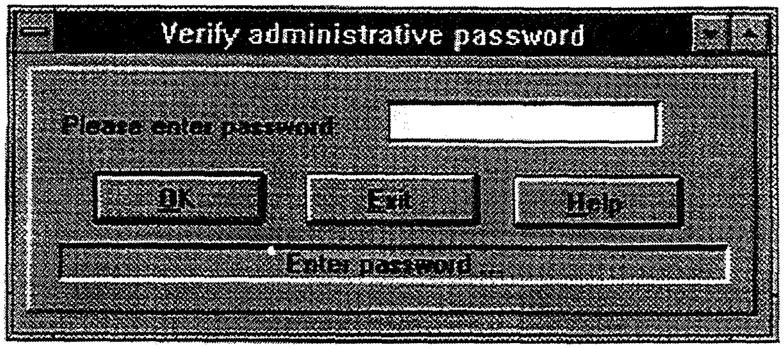


Fig. 4

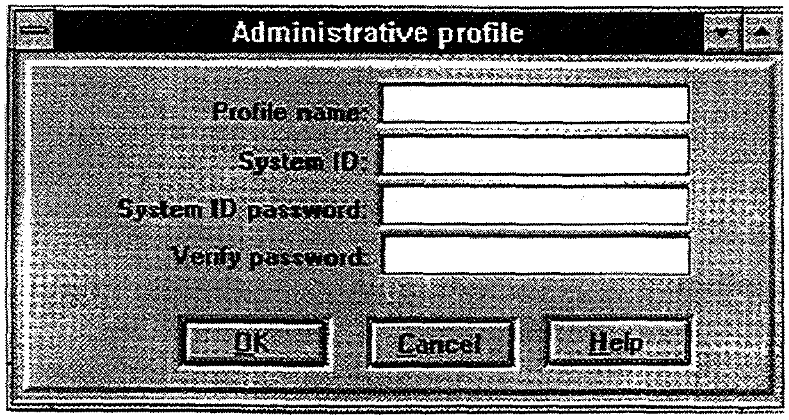


Fig. 5

2.1.5 Interface Speller

Interface Speller is a spell checking tool which reads all the terms used in widgets including menus, buttons, list boxes, combo boxes etc. throughout the interface and outputs terms that are not found in the dictionary. The spell checking operation is performed within the code and all the possible misspelled words are stored in a file. This file can be reviewed by the designer to detect possible misspelled and abbreviated words which may create confusion for the end users. The output is filtered through a file containing valid computer terms and default Visual Basic terms that may be detected as spelling errors by the dictionary. The tool detected few misspelled words, but many incomplete and abbreviated words such as "App", "Trans", "Ins" , "Opr" were found in all the test applications which are potentially confusing abbreviations.

2.1.6 Terminology Baskets

A terminology basket is a collection of computer task terms including their different tense formats which may be used as synonyms by the interface designers. Our goal is to construct different sets of terminology baskets by constructing our own computer thesaurus and then search for these baskets in every dialog box of the interface. The purpose of terminology baskets is to provide interface designers with feedback on misleading synonym computer terms, e.g. "Close", "Cancel", "End", "Exit", "Terminate", "Quit".
Our program reads an ASCII file containing the basket list. The baskets are sorted alphabetically and for each basket all the dialog boxes containing any of the basket terms are outputted. The list of baskets may be easily updated as more interfaces are analyzed in the future. Some of the idiosyncratic baskets were:

- Remove Removes Removed Removing Delete Deletes Deleted Deleting Clear Clears Cleared Clearing Purge Purges Purged Purging Cancel Cancels Canceled Canceling Refresh Refreshed
- Item Items Entry Entries Record Records Segment Segments Segmented Segmenting Field Fields
- Add Adds Added Adding Insert Inserts Inserted Inserting Create Creates Creating
- Message Messages Note Notes Letter Letters Comment Comments

Our basket browser revealed some interesting terminology anomalies after analyzing the large 130 dialog box interface that led to reconsideration of previous design. As shown below terms like "record", "segment", "field" and "item" were used in similar context in different dialog boxes. Other interesting anomalies included use of "start", "execute" and "run" for identical tasks in different dialogue boxes.

Basket: Entries, Entry, Field, Fields, Item, Itemized, Itemizing, Items
Record, Records, Segment, Segmented , Segmenting, Segments

Basket Term	**Form Containing the Basket Term**		
Field	search.cft		
Items	reconly.cft	reconly.cft	reconly.cft
	reconly.cft	sendrec.cft	sendrec.cft
	sendrec.cft	sendrec.cft	wastedef.cft
Record	ffadm.cft	profile.cft	
Segment	addr.cft	search.cft	

3. TESTING OUR EVALUATION TOOLS

Effectiveness of these consistency checking tools has been determined by evaluating two commercial prototype applications developed in Microsoft Visual Basic. These applications included a 139 and 30 dialog box GE Electronic Data Interchange Interface plus, a 51 and 29 dialog box University of Maryland, AT&T Teaching Theater Interface. Our testing method incorporates a sequence of steps beginning with applying the tools to the prototype application followed by analysis and review of the interface screen shots and outputs generated by our tools. Our evaluation tools were not created with reference to any particular test prototype and can evaluate any interface that is converted to the canonical format.

Our evaluation tools act as consistency patrollers reporting exceptions and anomalies, making interpretation easier for the developers. Both small and large applications had inconsistencies in use of proper typefaces and colors. Most of the screens used the same typefaces, but their were screens which used more than 5 different typefaces and seven different background or foreground colors, the reason being all of these applications had multiple designers working on the project. The large application had more terminology inconsistencies than the smaller applications., misleading synonyms were used for both labels and buttons for e.g. "File and Document", "Remove and Delete", "Add and Insert", "Search and Retrieve", "Item and Record", "Run and Execute". These terminology inconsistencies were detected by our terminology basket tool, which would have been left undetected otherwise. Button placement, button terminology and button capitalization inconsistencies were evident in all the applications. For eg. the most frequently used button set (OK, Cancel, Help) had inconsistent placement in every application. In some forms these buttons were placed on the top right of the dialog box, in others they were left aligned or right aligned or center aligned on the bottom of the dialog box. Sizes and relative position of buttons were also inconsistent. There were cases when no two OK buttons in an application had the same height and width. Button terminology inconsistencies like Cancel being replaced by Close and sometimes by Quit or Exit were also detected by our tools.

Test results and interpretations were shown to developers to elicit feedback and reactions. Terminology inconsistencies in the interface had the greatest impact on the developers, who after looking at the results modified the previously undiscovered synonym slips they had made. Another important concern was the use of so many different typefaces in a single form, previously undetected. There was an application in which 17 distinct typefaces were used in a 30 dialog box interface, developers went back to look at the application after seeing our results. The use of multiple typefaces and colors was due to multiple designers working on the application. Developers plan to stress more on these consistent terminology and layout issues in their internal guidelines in the future. Our consistency checking team is in the process of testing more complex commercial prototypes created by GE Information Services and other companies.

4. LIMITATIONS

Our evaluation tools are designed to aid the interface evaluation process by providing a compact overview of possible inconsistencies and anomalies on certain textual and spatial characteristics of the interface. The designer must decide what to do, if anything with these possible inconsistencies. Certain issues like efficiency in screen layout including proper placement of widgets on the dialog box, violation of any design constraints, use of inappropriate widgets types are not evaluated by our tools. Other evaluation methods, such as usability testing and heuristic evaluation, are needed to locate typical user interface design problems such as inappropriate metaphors, missing functionality, chaotic screen layouts, unexpected sequencing of screens, misleading menus, excessive demands on short-term memory, poor error messages, or inadequate help screens. Currently, the evaluation is limited to Visual Basic applications, but any experienced programmer can write a translator to convert interface form files created by other development tools to a canonical format read by our evaluation tools.

5. FUTURE DIRECTIONS

Currently, the printouts provided by our tools showing the possible anomalies and inconsistency patterns need to be compared manually with the interface dialog boxes. Checking back and forth between the printouts and dialog boxes to make corrections can be time consuming for large interfaces. It would be good to have these mini evaluation tools as interactive evaluation and modification tools. This would help developers to interactively make changes to the prototype while creating it rather than amassing printouts. In the future, we plan to incorporate the canonical format file translator and the evaluation tools together in Visual Basic. We also plan to diversify the metrics set of our evaluation tools to perform more detailed interface evaluation. We are currently working on writing a translator to convert the Visual C++ resource files into a Canonical format so that SHERLOCK can evaluate Visual C++ interfaces.

Acknowledgments

We appreciate the support for this project from GE Information Services and the Maryland Industrial Partnerships program. We thank Ren Stimart and David White at GE Information Services for their help in every part of this project. We thank Mun Yi and CS Chang at the University of Maryland for providing the test prototypes. We are grateful for draft comments from Richard Chimera, Catherine Plaisant, Anne Rose, Andrew Sears and Zhijun Zhang. We thank Eser Kandogan and Adil Rajput for providing programming assistance.

References

Ashlund, S. and Hix, D. (1992), "IDEAL: A tool to Enable User- Centred Design", *Proc. of CHI' 92 (Posters and short talk supplement to proceedings)* , ACM, New York, 119-120.

Bodart, F., Hennebert, A.-M., Leheureux, J.-M., and Vanderdonckt, J. (1994), "Towards a dynamic strategy for computer-aided visual placement", In Catarci, T., Costabile, M., Levialdi, S., and Santucci, G. (Editors), *Proc. Advanced Visual Interfaces Conference '94,* ACM Press, New York, 78-87.

Chimera, R. and Shneiderman, B. (1993), "User interface consistency: An evaluation of original and revised interfaces for a videodisk library", In Shneiderman, B. (Editor), *Sparks of Innovation in Human-Computer Interaction* , Ablex Publishers, Norwood, NJ, 259-271.

Comber, T. and Maltby, J. (1995), "Evaluating Usability of screen design with layout complexity", *Proc. of the CHISIG Annual Conference*, Melborne, Australia (in press, 6 pages)

Hix, D. and Hartson, H. R. (1993), *Developing User Interfaces: Ensuring Usability Through Product & Process*, John Wiley & Sons, New York, NY.

Kim, W. and Foley, J. (1993), "Providing high-level control and expert assistance in the user interface presentation design", *Proc. of CHI'93,* ACM, New York, 430-437.

Nielsen, J. and Molich, R., (1990) "Heuristic evaluation of user interfaces", *Proc. of CHI'90* , ACM, New York, 249-256.

Sears, A. (1993), "Layout Appropriateness: A metric for evaluating user interface widget layouts", *IEEE Transactions on Software Engineering* 19, 7, 707-719.

Sears, A. (1994), "Using automated metrics to design and evaluate user interfaces", DePaul University Dept of Computer Science Technical Report #94-002, Chicago, IL.

Shneiderman, B. (1992), *Designing the User Interface: Strategies for Effective Human-Computer Interaction: Second Edition,* Addison-Wesley Publ. Co., Reading, MA.

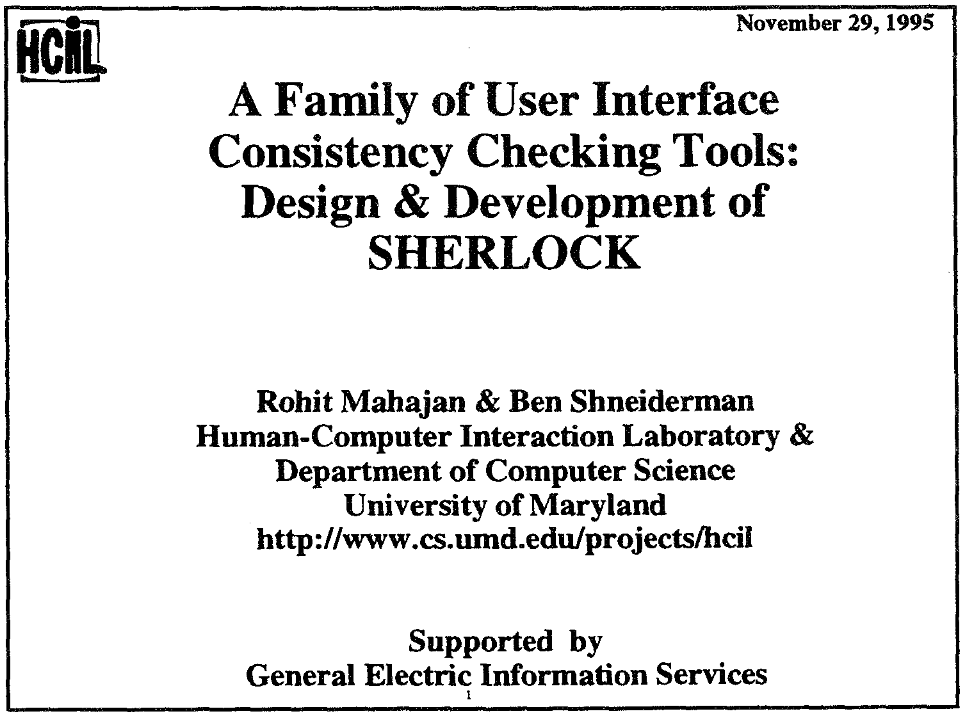
Shneiderman, B., Chimera, R., Jog, N., Stimart, R. and White, D. (1995), "Evaluating spatial and textual style of displays", *Proc. of Getting the Best from State-of the-Art Display Systems '95,* London.

Streveler, D. and Wasserman, A. (1987), "Quantitative measures of the spatial properties of screen designs", *Proc. of INTERACT '87,* Elsevier Science, Amsterdam, 125-133.

Tullis, T. S. (1988a), "Screen design", In Helander, M. (Editor), *Handbook of Human- Computer Interaction,* Elsevier Science, Amsterdam, The Netherlands, 377-411.

Tullis, T. S. (1988b), "A system for evaluating screen formats: Research and application", In Hartson, H. Rex and Hix, Hartson, *Advances in Human-Computer Interaction: Volume 2,* Ablex Publishing Corp., Norwood, NJ, 214-286.

Vanderdonckt, J. and Gillo, X. (1994), "Visual techniquės for traditional and multimedia layouts", In Catarci, T., Costabile, M., Levialdi, S. and Santucci, G. (Editors), *Proc. Advanced Visual Interfaces Conference '94,* ACM Press, New York, 95-104.

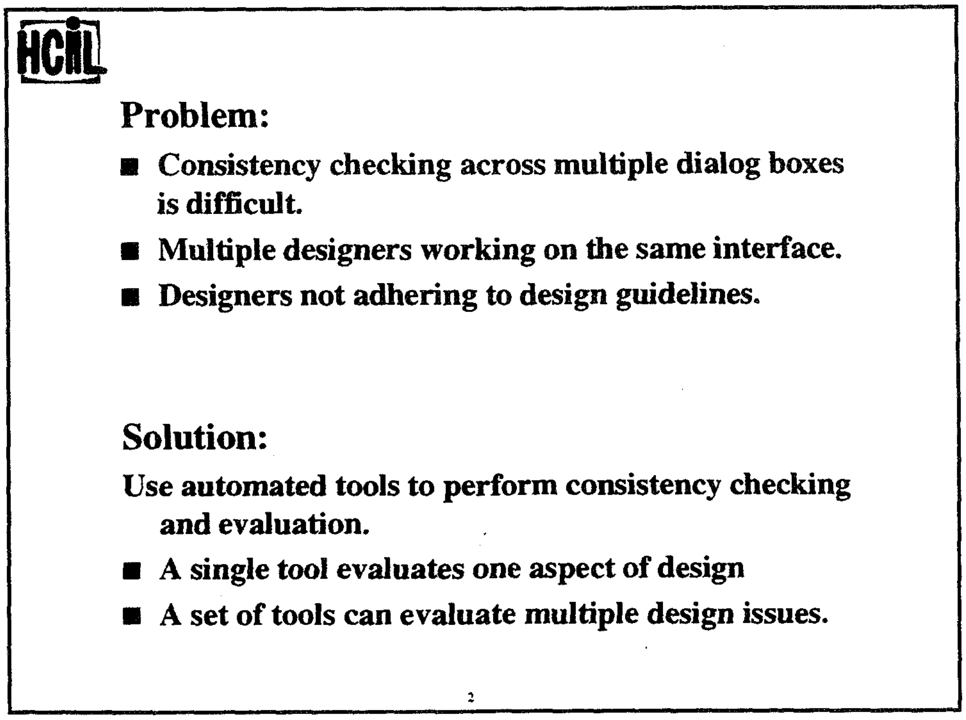


November 29, 1995

A Family of User Interface Consistency Checking Tools: Design & Development of SHERLOCK

Rohit Mahajan & Ben Shneiderman
Human-Computer Interaction Laboratory &
Department of Computer Science
University of Maryland
http://www.cs.umd.edu/projects/hcil

Supported by
General Electric Information Services



Problem:

- Consistency checking across multiple dialog boxes is difficult.
- Multiple designers working on the same interface.
- Designers not adhering to design guidelines.

Solution:

Use automated tools to perform consistency checking and evaluation.

- A single tool evaluates one aspect of design
- A set of tools can evaluate multiple design issues.

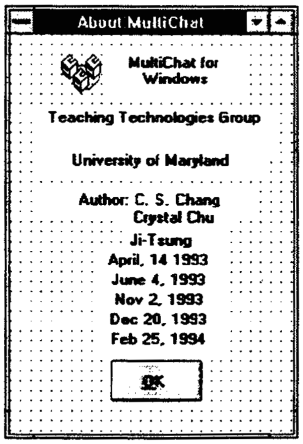
Our Evaluation Tools

Dialog Box Summary Table

- Compact overview of the spatial layout and visual properties of hundreds of dialog boxes.
- Each row of the table represents a single dialog box.
- Each column of the table represents a single metric.

3

Aspect Ratio (Height/Width)



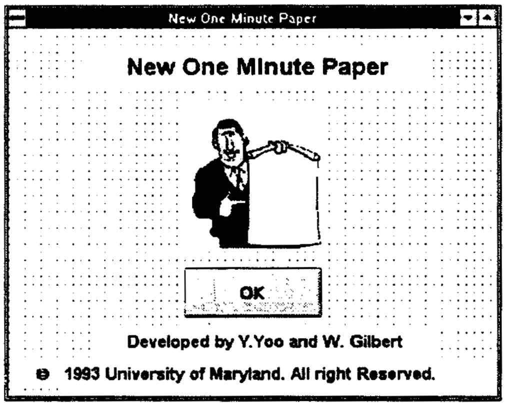
MultiChat for Windows

Teaching Technologies Group

University of Maryland

Author: C. S. Chang
Crystal Chu
Ji-Tsung
April, 14 1993
June 4, 1993
Nov 2, 1993
Dec 20, 1993
Feb 25, 1994

About_M.frm



New One MInute Paper

Developed by Y.Yoo and W. Gilbert

© 1993 University of Maryland. All right Reserved.

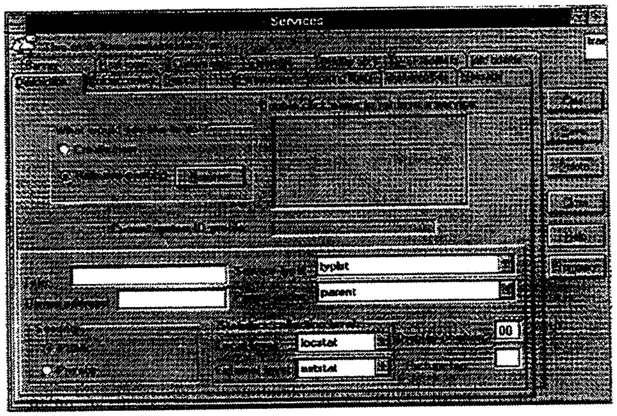
About_N.frm

Aspect Ratio = 1.50

Aspect Ratio = 0.84

4

Widget Totals



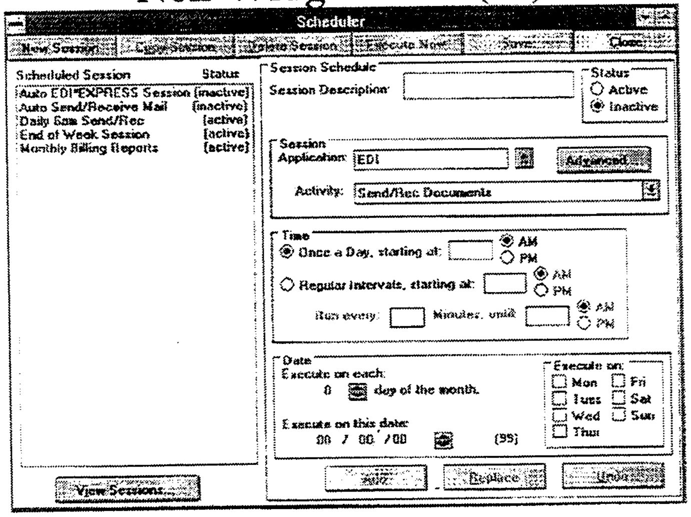
Servinp.cft

Widget Totals:

- Top Level = 14
- All = 215

5

Non-Widget Area (%)



Scheduled Session	Status
Auto EDI*EXPRESS Session	[inactive]
Auto Send/Receive Mail	[inactive]
Daily 8am Send/Rec	[active]
End of Week Session	[active]
Monthly Billing Reports	[active]

Session Schedule

Session Description:

Status: ○ Active ● Inactive

Session

Application: EDI

Activity: Send/Rec Documents

Time

● Once a Day, starting at: ● AM ○ PM

○ Regular intervals, starting at: ● AM ○ PM

Run every: Minutes, until ● AM ○ PM

Date

Execute on each: 0 day of the month.

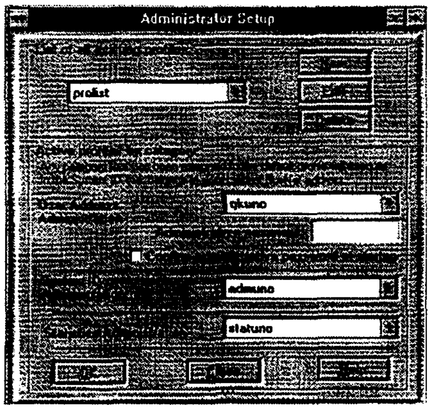
Execute on this date: 00 / 00 / 00 (95)

Execute on: □ Mon □ Tues □ Wed □ Thur □ Fri □ Sat □ Sun

Non-Widget Area = 5%

6

Widget Density



prolist

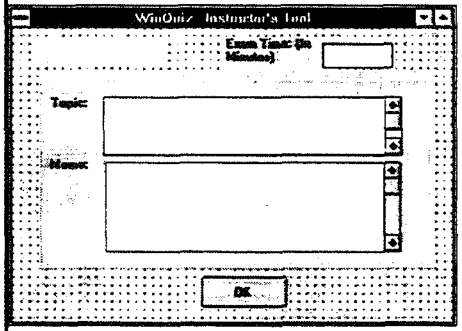
qkuno

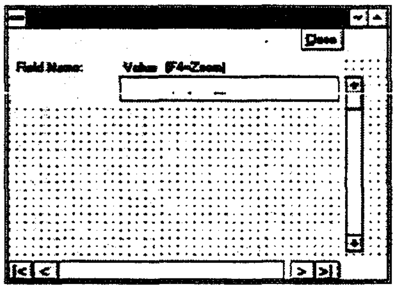
admuno

statuno

Widget Density = 206

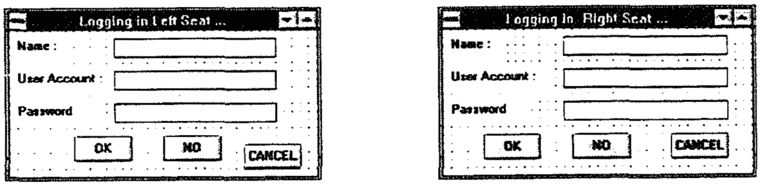
Margins (in pixels)





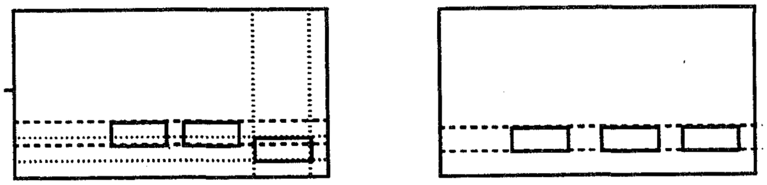
Left Margin = 23	Right Margin = 18	Left Margin = 0	Right Margin = 21
Top Margin = 16	Bottom Margin = 16	Top Margin = 0	Bottom Margin = 1

Button Gridedness

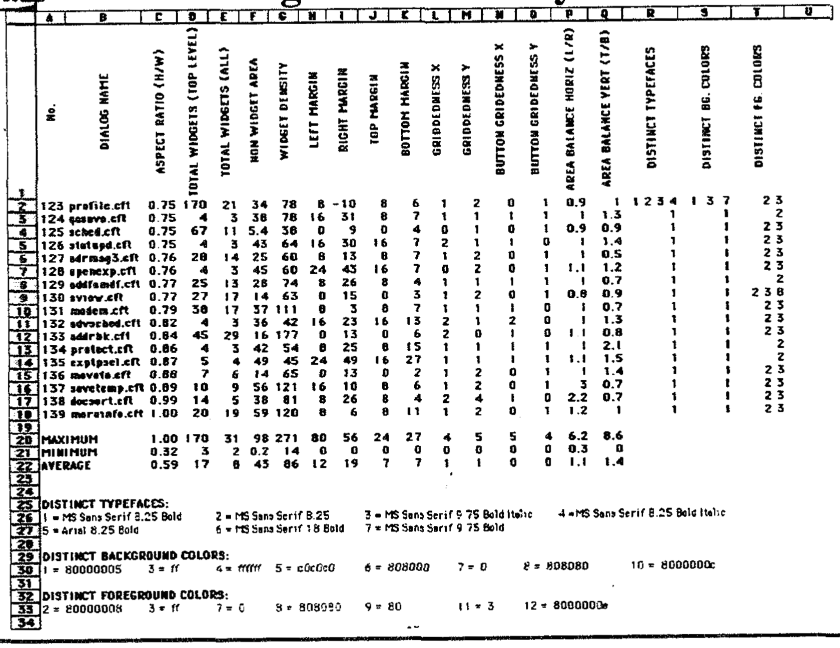


X-Gridedness = 1
Y-Gridedness = 2

X-Gridedness = 0
Y-Gridedness = 1



Dialog Box Summary Table

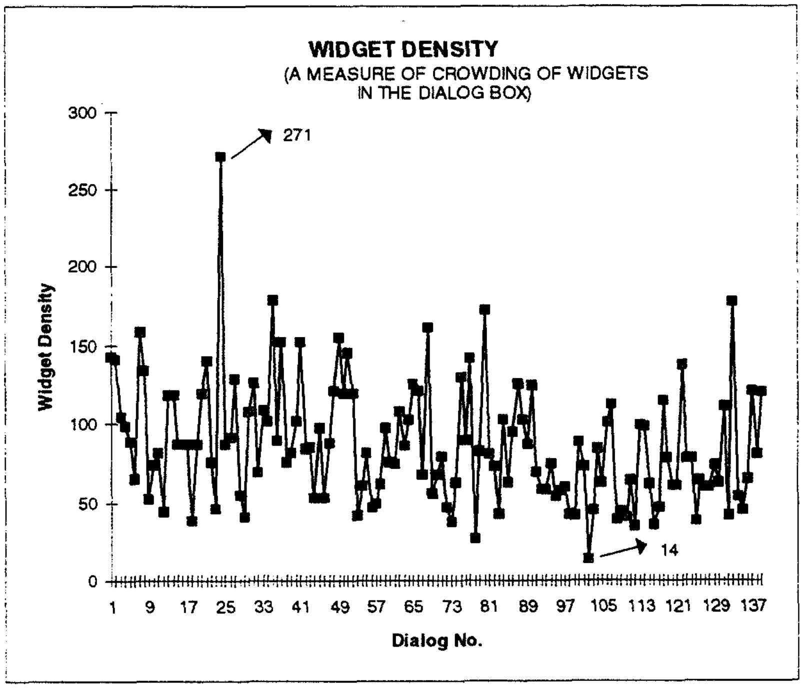


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	No.	DIALOG NAME	ASPECT RATIO (H/W)	TOTAL WIDGETS (TOP LEVEL)	TOTAL WIDGETS (ALL)	NON WIDGET AREA	WIDGET DENSITY	LEFT MARGIN	RIGHT MARGIN	TOP MARGIN	BOTTOM MARGIN	GRIDDEDNESS X	GRIDDEDNESS Y	BUTTON GRIDEDNESS X	BUTTON GRIDEDNESS Y	AREA BALANCE HORIZ (L/R)	AREA BALANCE VERT (T/B)	DISTINCT TYPEFACES	DISTINCT BG. COLORS	DISTINCT FG. COLORS	
2	123	profile.cft	0.75	170	21	34	78	8	-10	8	6	1	2	0	1	0.9	1	1 2 3 4	1 3 7	2 3	
3	124	qosavo.cft	0.75	4	3	38	78	16	31	8	7	1	1	1	1	1	1.3	1	1	2	
4	125	sched.cft	0.75	67	11	5.4	38	0	9	0	4	0	1	0	1	0.9	0.9	1	1	2 3	
5	126	statupd.cft	0.75	4	3	43	64	16	30	16	7	2	1	1	0	1	1.4	1	1	2 3	
6	127	adrmag3.cft	0.76	28	14	25	60	8	13	8	7	1	2	0	1	1	0.5	1	1	2 3	
7	128	openexp.cft	0.76	4	3	45	60	24	43	16	7	0	2	0	1	1.1	1.2	1	1	2 3	
8	129	addfamdf.cft	0.77	25	13	28	74	8	26	8	4	1	1	1	1	1	0.7	1	1	2	
9	130	sview.cft	0.77	27	17	14	63	0	15	0	3	1	2	0	1	0.8	0.9	1	1	2 3 8	
10	131	modem.cft	0.79	38	17	37	111	8	3	8	7	1	1	1	0	1	0.7	1	1	2 3	
11	132	advsched.cft	0.82	4	3	36	42	16	23	16	13	2	1	2	0	1	1.3	1	1	2 3	
12	133	addrbk.cft	0.84	45	29	16	177	0	13	0	6	2	0	1	0	1.1	0.8	1	1	2 3	
13	134	protect.cft	0.86	4	3	42	54	8	25	8	15	1	1	1	1	1	2.1	1	1	2	
14	135	explpsel.cft	0.87	5	4	49	45	24	49	16	27	1	1	1	1	1.1	1.5	1	1	2	
15	136	movefo.cft	0.88	7	6	14	65	0	13	0	2	1	2	0	1	1	1.4	1	1	2 3	
16	137	savetemp.cft	0.89	10	9	56	121	16	10	8	6	1	2	0	1	3	0.7	1	1	2 3	
17	138	docsort.cft	0.99	14	5	38	81	8	26	8	4	2	4	1	0	2.2	0.7	1	1	2 3	
18	139	moreinfo.cft	1.00	20	19	59	120	8	6	8	11	1	2	0	1	1.2	1	1	1	2 3	
19																					
20	MAXIMUM		1.00	170	31	98	271	80	56	24	27	4	5	5	4	6.2	8.6				
21	MINIMUM		0.32	3	2	0.2	14	0	0	0	0	0	0	0	0	0.3	0				
22	AVERAGE		0.59	17	8	43	86	12	19	7	7	1	1	0	0	1.1	1.4				

DISTINCT TYPEFACES:
1 = MS Sans Serif 8.25 Bold   2 = MS Sans Serif 8.25   3 = MS Sans Serif 9 75 Bold Italic   4 = MS Sans Serif 8.25 Bold Italic
5 = Arial 8.25 Bold   6 = MS Sans Serif 18 Bold   7 = MS Sans Serif 9 75 Bold

DISTINCT BACKGROUND COLORS:
1 = 80000005   3 = ff   4 = ffffff   5 = c0c0c0   6 = 808000   7 = 0   8 = 808080   10 = 8000000c

DISTINCT FOREGROUND COLORS:
2 = 80000008   3 = ff   7 = 0   8 = 808080   9 = 80   11 = 3   12 = 8000000e



Concordance & Terminology Checking Tools

- Interface Concordance
- Button Concordance
- Button Layout Table
- Interface Speller
- Terminology Baskets

12

Button Concordance

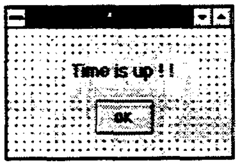
Button	Form Containing The Button	Button Typeface	Button Fg_Color	Button (H,W)
EXIT	delete.frm.cft	1	1	41, 97
Exit	attapp94.frm.cft	1	1	56, 120
	winstat.frm.cft	3	1	49, 113
OK	find.frm	1	1	24, 112
	grid3.frm.cft	1	1	24, 68
	help.frm.cft	1	2	32, 72
	list2.frm.cft	1	1	29, 92
Ok	form3.frm.cft	1	2	33, 105

Distinct Typefaces in Buttons:
1= MS Sans Serif 8.25 Bold
2= MS Sanss Serif 12 Bold
3= MS Serif 12 Bold

Distinct Fg_Colors in Buttons:
1= ffffffff80000005
2= 404040

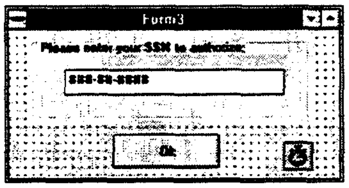
13

Button Concordance



Time is up ! !

timeup2.frm.cft

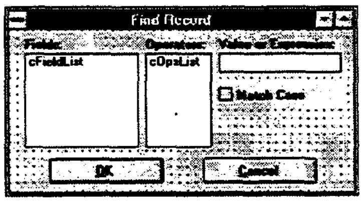


Form3

Please enter your SSN to authorize:

888-88-8888

form3.frm.cft



Find Record

find.frm.cft

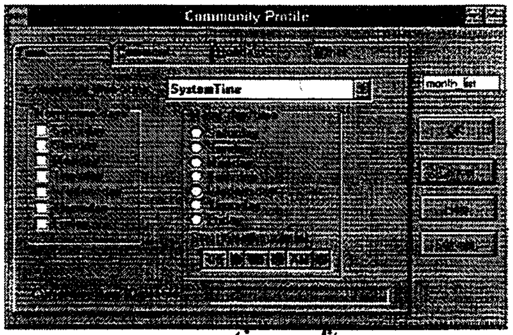
14

Button Layout Table

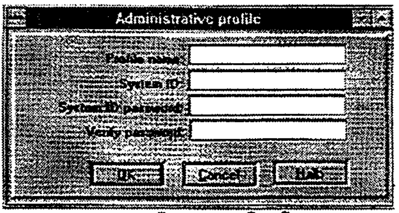
Form Name	OK	Cancel		Exit		Help	
	(H,W)	(H,W)	Rel. Pos.	(H,W)	Rel. Pos.	(H,W)	Rel. Pos.
admprof.cft	22,68	22,68	x+16, y			22,68	x+98, y
checkpsw.cft	25,82			25,82	x+18, y	25,82	x+116, y+1
nbatch.cft	25,62	25,62	x-1, y+20			25,62	x, y+66
systinp.cft	26,72	26,72	x+1, y+13			25,73	x+2, y+48

15

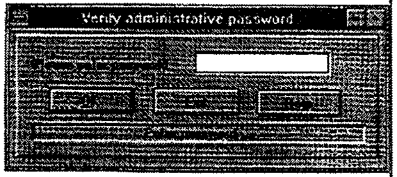
Button Layout Table



systimp.cft

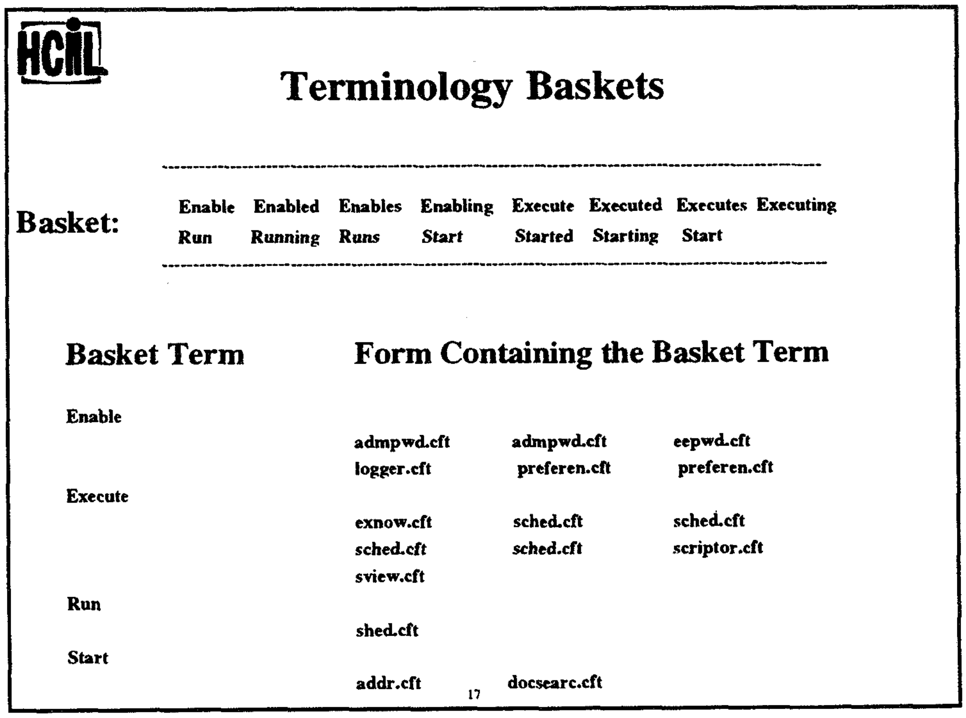


admprof.cft



checkpsw.cft

16

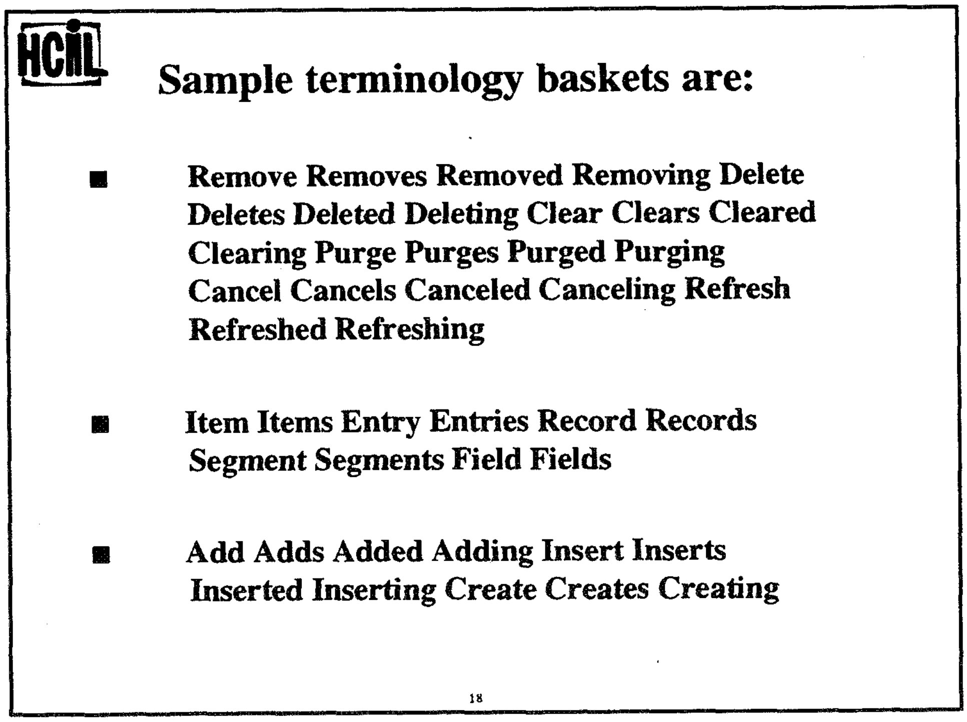


Terminology Baskets

Basket:
Enable Enabled Enables Enabling Execute Executed Executes Executing
Run Running Runs Start Started Starting Start

Basket Term	Form Containing the Basket Term		
Enable	admpwd.cft	admpwd.cft	eepwd.cft
	logger.cft	preferen.cft	preferen.cft
Execute	exnow.cft	sched.cft	sched.cft
	sched.cft	sched.cft	scriptor.cft
	sview.cft		
Run	shed.cft		
Start	addr.cft	docsearc.cft	

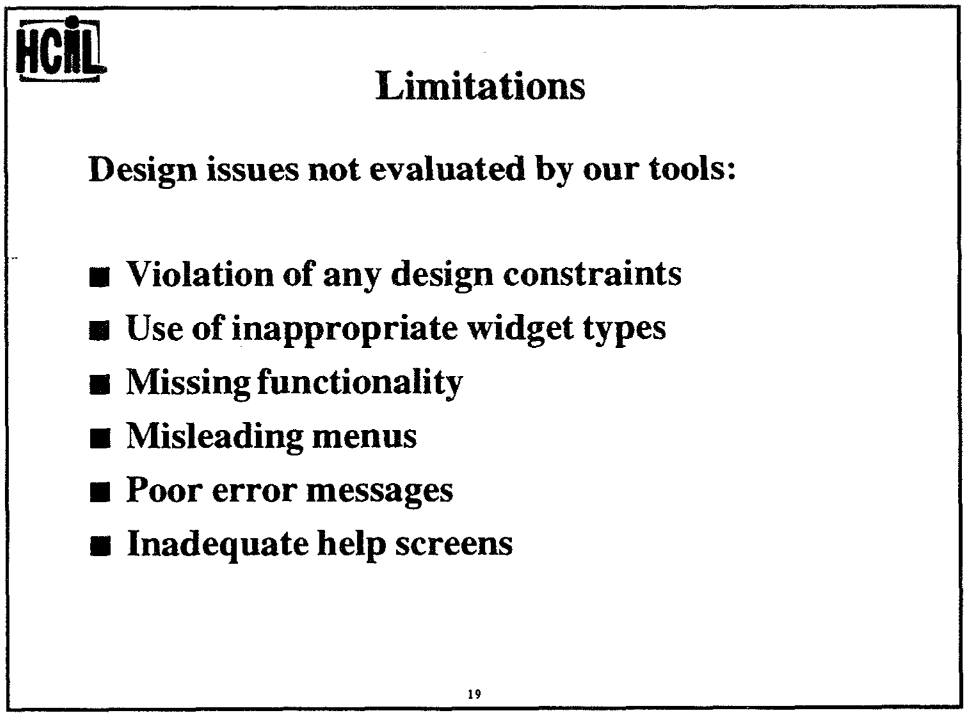
17



Sample terminology baskets are:

- Remove Removes Removed Removing Delete Deletes Deleted Deleting Clear Clears Cleared Clearing Purge Purges Purged Purging Cancel Cancels Canceled Canceling Refresh Refreshed Refreshing
- Item Items Entry Entries Record Records Segment Segments Field Fields
- Add Adds Added Adding Insert Inserts Inserted Inserting Create Creates Creating

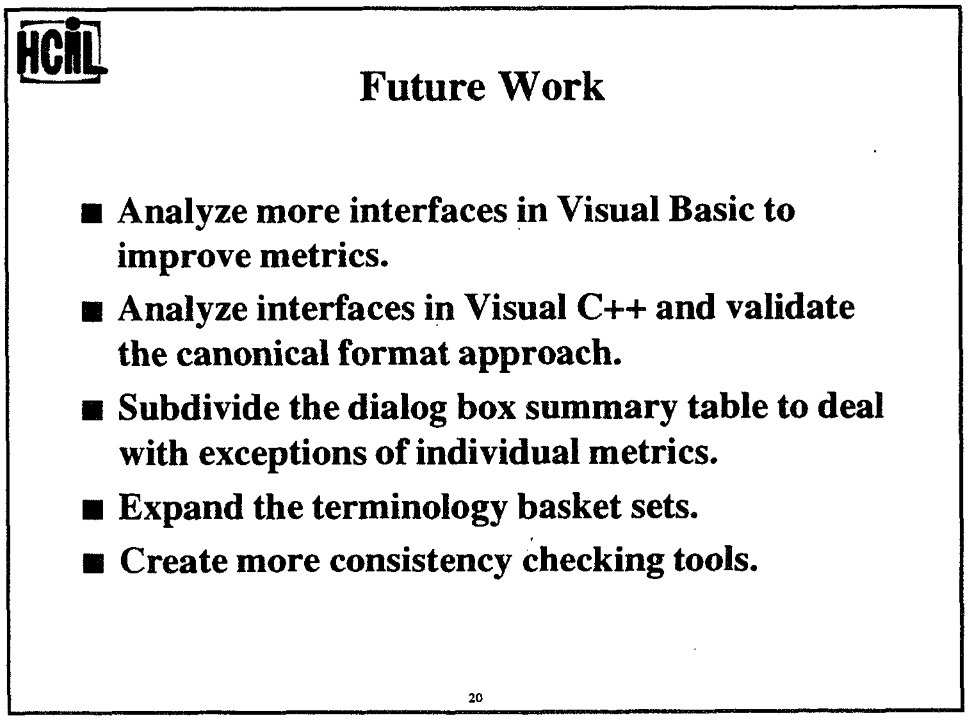
18



Limitations

Design issues not evaluated by our tools:

- Violation of any design constraints
- Use of inappropriate widget types
- Missing functionality
- Misleading menus
- Poor error messages
- Inadequate help screens



Future Work

- Analyze more interfaces in Visual Basic to improve metrics.
- Analyze interfaces in Visual C++ and validate the canonical format approach.
- Subdivide the dialog box summary table to deal with exceptions of individual metrics.
- Expand the terminology basket sets.
- Create more consistency checking tools.

A COTS Selection Method and Experiences of Its Use*

Jyrki Kontio[1], Show-Fune Chen[2], Kevin Limperos[2],
Roseanne Tesoriero[1], Gianluigi Caldiera[1], Mike Deutsch[2]

[1] University of Maryland
Department of Computer Science
A.V.Williams Building
College Park, MD 20742, U.S.A.
[jkontio, roseanne, gcaldiera]@cs.umd.edu

[2] Hughes Information Technology Corporation
1616A McCormick Dr.
Landover, MD 20785-5372, U.S.A.
[schen, klimpero, miked]@eos.hitc.com

Abstract:

This paper presents the OTSO method for reusable component selection. The OTSO method has been developed to provide a basis for evaluating and selecting reusable components for software development. The main characteristics of the OTSO method include (i) a well-defined, documented process, (ii) hierarchical and detailed evaluation criteria decomposition and definition, (iii) a model for making alternatives comparable in terms of cost and added value they produce, and (iv) use of appropriate techniques for consolidating evaluation data.

The OTSO method has been evaluated in two real-world case studies. The case studies indicated that a well-defined process allows the selection process to take place efficiently, the overhead of formal criteria definition is marginal, and the use of different data consolidation methods may influence the results.

1. Introduction

Reuse has been considered an important solution to many of the problems in software development. It has been claimed to be important in improving productivity and quality of software development [2,7,16,24,30,33] and significant benefits have been reported by many organizations [14,23]. A large volume of research has produced several useful tools to support reuse [16,30,34] but it is widely believed that successful reuse is not only dependent on technical issues, it also requires the solving of organizational, motivational and legal issues [4,5,33,35,36]. It has been argued that an important characteristic of the infrastructure supporting reuse is the existence of a "marketplace" that both provides access to reuse producers and consumers as well as provides a mechanism to transfer benefits between the parties [8,22,23,37].

* This work has been sponsored by the Hughes Information Technology Corporation.

Many organizations have implemented systematic reuse programs [14] which have resulted in in-house libraries of reusable components. The increased commercial offering of embeddable software components, standardization of basic software environments (e.g., MS-Windows, UNIX), and popularization of Internet have resulted in a new situation for reusable software consumers: there are many more accessible, potential reuse candidates. Given the high interest in reuse and motivation to the use of commercially available software, many software development projects include the evaluation and selection of reusable components as an important activity in the project, with a high potential impact on the product and project objectives. According to our observations in many organizations, the selection process typically is not defined, each project finds its own approach to it, often under schedule pressure, and there are no mechanisms to learn from previous selection cases. Yet the selection of the right reusable component is often a non-trivial task and requires careful consideration of multiple criteria and careful balancing between application requirements, technical characteristics and financial issues. It seems that there is a lot of potential for non-optimal or inconsistent software reuse decisions.

However, the issues and problems associated with the selection of suitable reusable components have rarely addressed in the reuse community. Poulin et al. present an overall selection process [23] and include some general criteria for assessing the suitability of reuse candidate [32]. Some general criteria have been proposed to help in the search of potential reusable components [24,25]. Boloix and Robillard recently presented a general framework for assessing the software product, process and their impact on the organization [9]. However, little of this work is specific to externally developed, off-the-shelf (COTS[1]), software selection and the issues of how to define the evaluation criteria are not addressed. Furthermore, most of the reusable component literature does not seem to emphasize the sensitivity of such criteria to each situation.

We have developed a method that addresses the selection process of packaged, reusable off-the-shelf software. The method, called OTSO[2], supports the search, evaluation and selection of reusable software and provides specific techniques for defining the evaluation criteria, comparing the costs and benefits of alternatives, and consolidating the evaluation results for decision making.

We have applied the OTSO method in two case studies that are referred to in this paper. These case studies indicate that the method is feasible and has a low overhead. It also seems that the method results in efficient and consistent evaluations and increases decision makers' confidence in evaluation results.

This paper presents the OTSO method and its underlying principles. We have reported more details about the method and its usage experiences separately [17-20].

[1] COTS stands for "commercial off-the-shelf". This term is frequently used to refer to software packages that have been developed or are suitable for reuse. In this paper the term refers to all off-the-shelf software, regardless of its origin (commercial or in-house).
[2] OTSO stands for Off-The-Shelf Option. The OTSO method represents a systematic approach to evaluate such an option.

2. The OTSO Method

The OTSO method was developed to facilitate a systematic, repeatable and requirements-driven COTS software selection process. The main principles of the OTSO method are the following:

- a well-defined, systematic process that covers the whole reusable component selection process,
- a systematic method for deriving detailed COTS software evaluation criteria from reuse goals
- a method for estimating the relative effort or cost-benefits of different alternatives,
- a method for comparing the "non-financial" aspects of alternatives, including situations involving multiple criteria, and

The overall phases of COTS software selection are presented in Figure 1. The horizontal axis in Figure 1 represents the progress of the evaluation (i.e., time) and vertical axis the number of alternatives considered at each phase. Starting by the *search* phase, the number of possible alternatives may grow quite rapidly. The most potential candidates will need to be sorted out (*screening*) to pick the ones that can be evaluated in more detail with the resources available. Detailed *evaluation* of a limited number of alternatives determines how well each of the alternatives meets the evaluation criteria. These results are systematically documented. We have separated out the *analysis* phase to emphasize the importance of interpreting evaluation data. Sometimes it may be possible to make straight-forward conclusions if one of the alternatives is clearly superior to others. However, in most cases it is necessary to use systematic multiple criteria decision making techniques to arrive at a decision. Based on the decisions made, typically one of the alternatives is selected and *deployed.* Finally, in order to improve the selection process and to provide feedback on potential further reuse of the component, it is necessary to *assess* the success of the reuse component used in a project.

Figure 1 presents a high level, sequential view of the OTSO selection process. In Figure 2 we have presented a more realistic and detailed view of the OTSO process, using a data flow diagram notation. Figure 2 highlights the central role of evaluation criteria definition. In

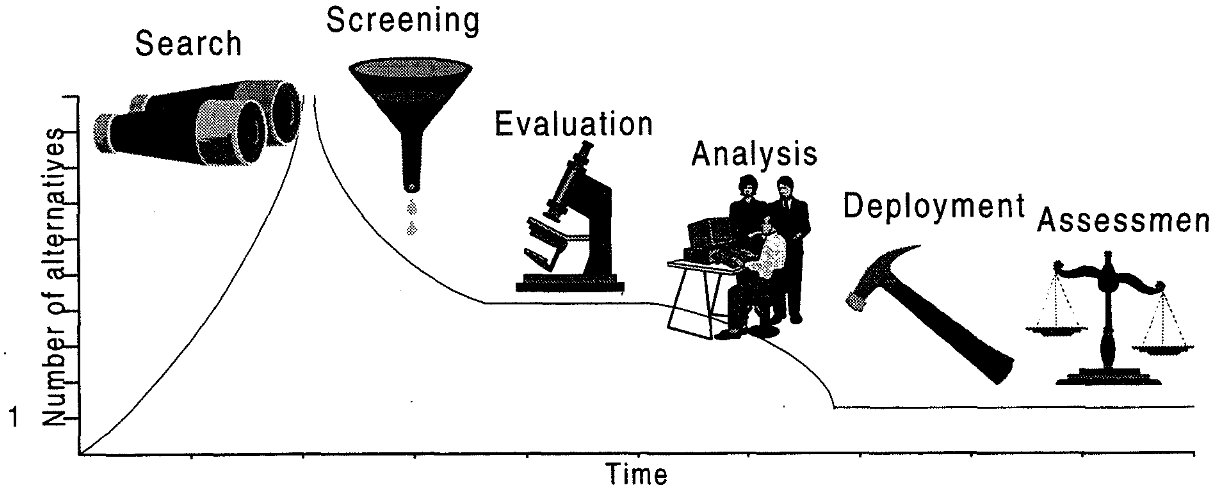


Figure 1: The phases in COTS selection

our method, the evaluation criteria are gradually defined as selection process progresses. The evaluation criteria are derived from reuse goals and factors that influence these goals [17].

2.1 Evaluation Criteria Definition

The evaluation criteria are formally defined so that the evaluation of alternatives can be conducted efficiently and consistently. We have defined a template that can be used for such definition [17,18]. As a minimum, the evaluation attribute definitions should include a detailed description of the attribute, its rationale, as well as the scale and measurement unit used.

The evaluation criteria definition process essentially decomposes the requirements for the COTS software into a hierarchical criteria set. Each branch in this hierarchy ends in an *evaluation attribute*: a well-defined measurement or a piece of information that will be determined during evaluation. This hierarchical decomposition principle has been

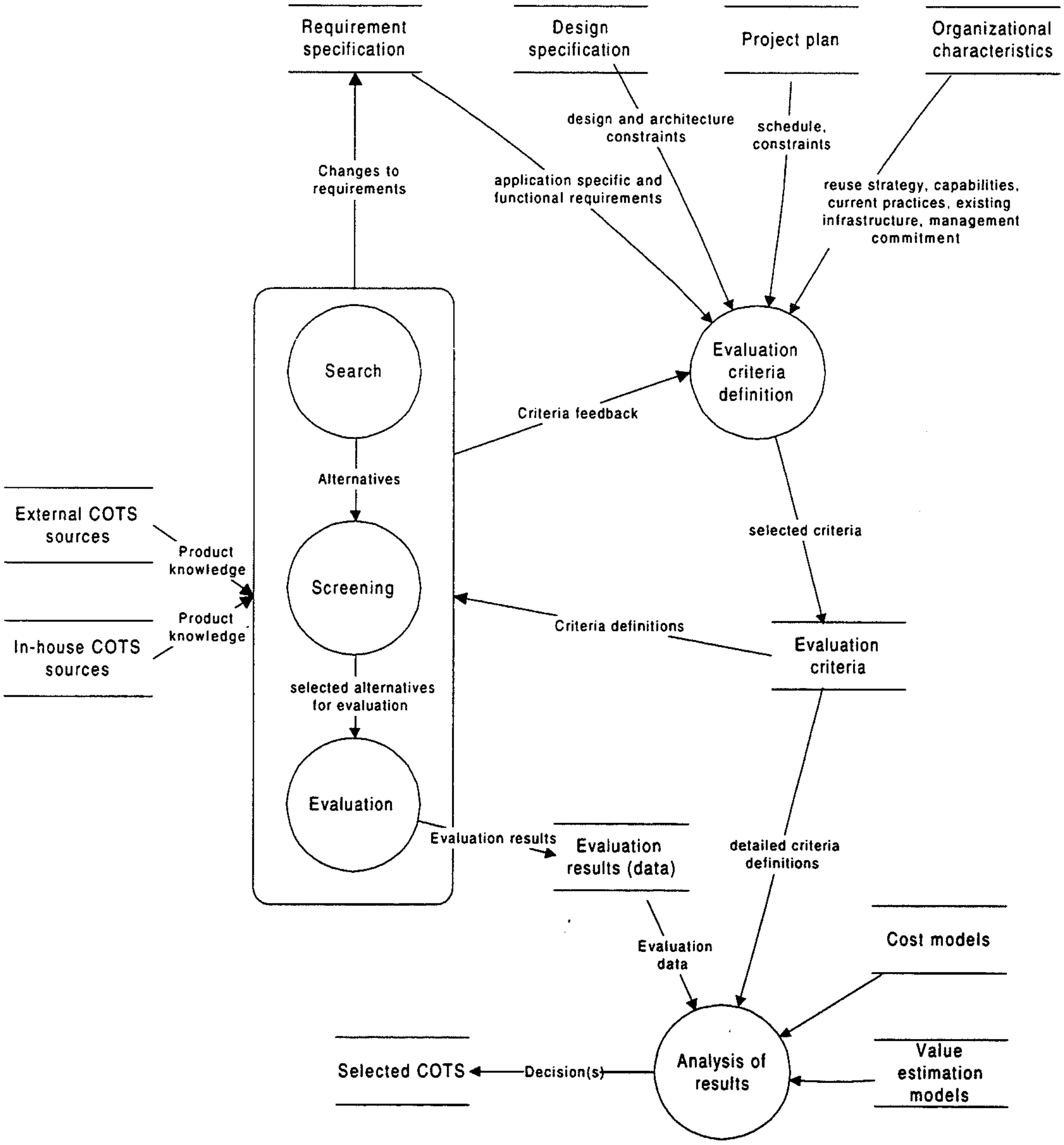


Figure 2: The OTSO selection process

derived from Basili's GQM [3,6] and Saaty's approach [27]. The evaluation attributes should have clear operational definitions so that consistency can be maintained during evaluation. The decomposition principles have been described in a separate technical report [17].

It is possible to identify four different subprocesses in the definition of evaluation criteria search criteria definition, definition of the baseline, detailed evaluation criteria definition, and weighting of criteria. Figure 3 presents a graphical representation of these processes.

First, when the available alternatives are searched and surveyed it is necessary to define the main search criteria and the information that needs to be collected for each alternative. The search criteria is typically based on the required main functionality (e.g., "visualization of earth's surface" or "hypertext browser") and some key constraints (e.g., "must run on Unix and MS-Windows" or "cost must be less than $X"). As far as the main functionality is concerned, an effective way to communicate such requirements is to use an existing product as a reference point, i.e., defining the functionality search criteria as "look for COTS products that are similar to our prototype".

It is enough to define the search criteria broadly so that the search is not unnecessarily limited by too many constraints. The reuse strategy and application requirements are used as the main input in the definition of this criteria. In Figure 3 the search criteria definition and actual search are presented as separate processes.

The screening process uses the criteria and determines the "qualifying thresholds", which are in deciding which alternatives are selected for closer evaluation. These threshold values will be documented together with the criteria definitions.

The definition of the baseline criteria set is essential for cost estimates and for conducting qualitative ranking of alternatives, as we will discuss later in this document. This can be done in parallel with the detailed evaluation criteria definition.

The search criteria, however, often is not detailed enough to act as a basis for detailed technical evaluation. Therefore, the criteria will need to be refined and formalized before initiating the technical evaluation. The evaluation criteria for the search of alternatives do not need to be very detailed or formally defined. However, as we discussed earlier, there must be detailed and unambiguous definitions for the criteria before detailed technical evaluation can be carried out. Without such definitions it is difficult to conduct a consistent and systematic evaluation, let alone consolidate the evaluation results for decision making. We have defined a template for criteria test definition that helps in defining criteria and tests in adequate detail [17,18].

2.2 Search

The search in the selection process attempts to identify and find all potential candidates for reuse. The search is driven by the guidelines and criteria defined in the criteria definition process. By its nature, search is an opportunistic process and it is not meaningful to define it formally in detail. However, some guidelines about the main issues involved in the search can be presented.

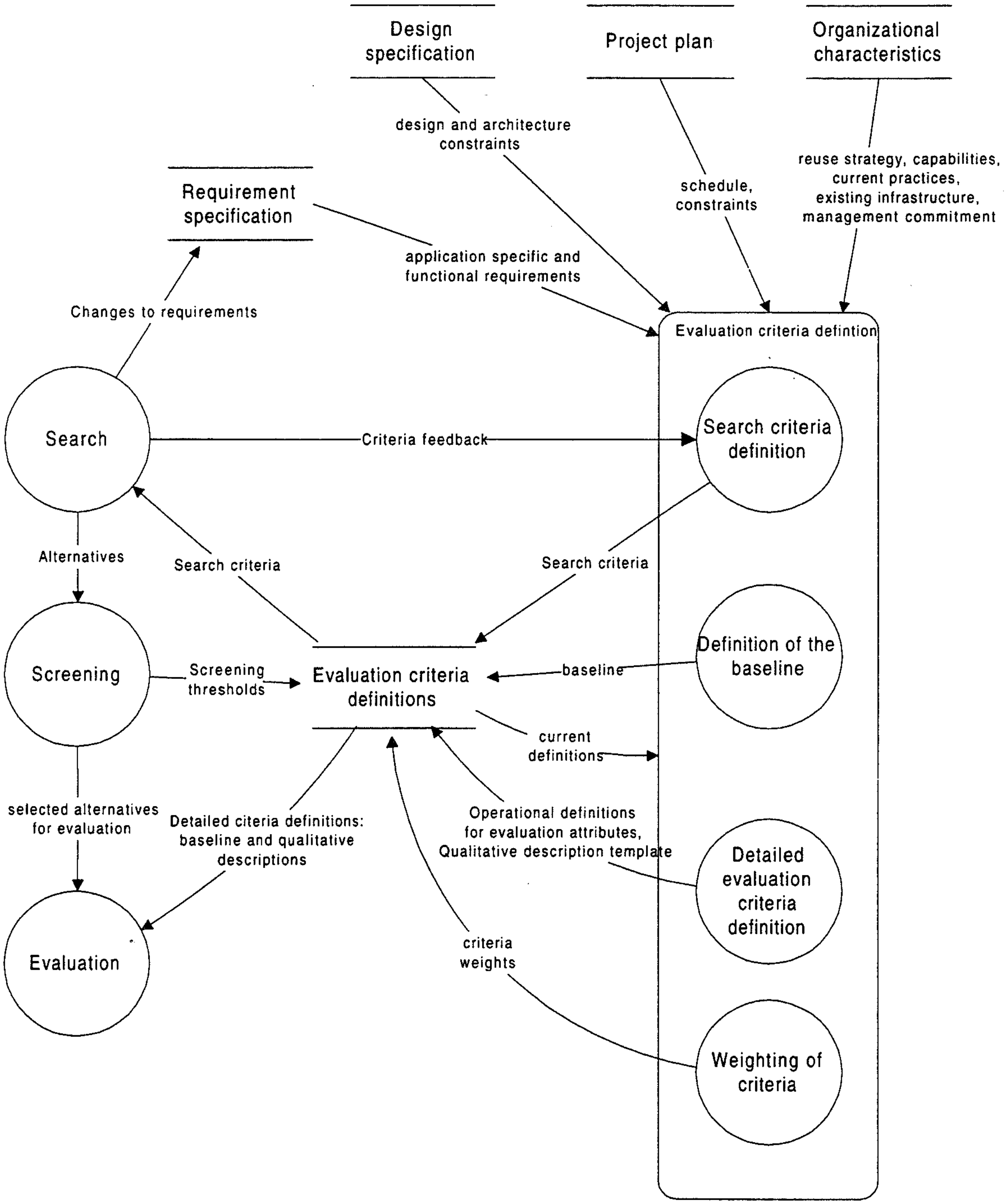

Design specification
Project plan
Organizational characteristics
design and architecture constraints
schedule, constraints
reuse strategy, capabilities, current practices, existing infrastructure, management commitment
Requirement specification
application specific and functional requirements
Changes to requirements
Evaluation criteria defintion
Search
Criteria feedback
Search criteria definition
Alternatives
Search criteria
Search criteria
Screening
Screening thresholds
Evaluation criteria definitions
baseline
Definition of the baseline
current definitions
selected alternatives for evaluation
Detailed citeria definitions: baseline and qualitative descriptions
Operational definitions for evaluation attributes, Qualitative description template
Detailed evaluation criteria definition
Evaluation
criteria weights
Weighting of criteria

Figure 3: Evaluation criteria definition process[3]

It is important to use several sources, or leads, of information in the search process. Relying on a single source limits the search space drastically. If the search for COTS software is repeated often in an organization, it is a good idea to document the possible sources well so that access to these is as easy as possible. Typical sources are described in the following.

[3] Note that the Figure 3 is a refinement of Figure 2.

In-house reuse libraries: an organization may have an internal library of components that have been developed for reuse. This internal reuse library should be used to determine whether any suitable components exist.

Internet and World Wide Web: they contain large amounts of up-to-date information on most commercial and shareware software products. Some search facilities may be used, e.g., the following may provide good leads:

- Yahoo -- http://www.yahoo.com/
- Lycos -- http://www.lycos.com/
- InfoSeek -- http://www.infoseek.com/
- CUI W3 Catalog -- http://cuiwww.unige.ch/w3catalog
- WWW Virtual Library -- htp://www.w3.org/hypertext/DataSources/bySubject/Overview.html

Magazines and journals: there are several magazines that contain reviews of products and large amounts of advertisements. Many of these are dedicated to the type of platform (e.g., Mac or MS-Windows), given technology (e.g., object oriented programming, user interfaces or databases) or application area.

Trade shows and conferences: many conferences include extensive vendor exhibitions where it is possible to see several products at the same time, ask detailed questions and order for more information.

Vendors: once some vendors have been recognized, one of the best ways to identify the most important competitors is to ask the vendors directly ("what are your main competitors and how is your product different from them?").

Colleagues, experts and consultants: it is important to utilize the network of people that may have been exposed to reuse candidates.

Other organizations: other organizations may have developed software that has the required features and functionality. They may have internal reuse programs that may make it easy to access a large amount of reuse candidates. Even when there is no reuse library and there may not be components that have been developed for reuse, it may be possible to identify similar applications and define joint development efforts. The potential for such sharing of reusable components is particularly promising in the government domain, as the proprietary and competitive issues are not as big of a problem as they may in industry.

The search process can be initiated as soon as the main features of the required component have been defined. In other words, the entry criterion is: main features for the reuse candidates have been defined.

One main challenge in the search is the difficulty of deciding when to stop the search: how do you know that you have searched enough and found all the relevant alternatives ? A simple strategy for ending the search is to use several sources in the search, conduct the search in small increments (e.g., a few days at a time) and review the frequency of discovering new alternatives at each increment. When the all sources have resulted in more or less the same set of alternatives and new alternatives have not appeared for a while, there is a reason to believe that the marginal benefits of additional searches are low.

The note that the search process can also influence both the requirements defined for the whole system and the evaluation criteria. It is quite possible that when new tools are encountered, they trigger new ideas about the possible functionality in the application. This is an important feedback mechanism that can be used to enhance the development process and user satisfaction.

2.3 Screening

The objective of the screening process is to decide which alternatives should be selected for more detailed evaluation. In most cases the results of the search process are too general to be taken as the basis for the COTS software reuse decision. Evaluating and analyzing all the relevant characteristics of any one alternative takes a non-trivial amount of time, typically more than the organization has available for evaluating all of the alternatives. Therefore, it is both necessary and cost-effective to select the most promising candidates for detailed evaluation.

Screening is based on the same criteria that was used in the search process. In screening, the "qualifying thresholds" are defined. In other words, the criteria and rationale for selecting alternatives for detailed evaluation is defined and documented.

The screening process can be initiated as soon as there is at least one relevant alternative to consider. This may be, in fact, a necessary way to shorten the overall duration of the selection process: arrangements for obtaining copies of tools for evaluation can be initiated as soon as decisions are made. While incremental screening decisions may result premature decisions and it may be theoretically biased, in practice it may be a very important technique to reduce the overall duration of the process.

Screening is considered to be complete when evaluation alternatives selected and evaluation tasks have been assigned. Note that when the screening is done, the process may need to reactivated if new alternatives are discovered.

2.4 Evaluation

The objective of the evaluation process is to evaluate the selected alternatives by the evaluation criteria and document evaluation results. Evaluation produces data on how well each alternative meets the criteria defined.

Evaluation includes the practical arrangements of obtaining copies of the tools to be evaluated, installing them, learning to use them, studying their features and assessing them against the criteria. It is important to point out that there can be considerable time delays in the evaluation process. Procurement for obtaining legitimate copies of tools may take time, as well as shipping and handling, there may be significant installation problems due to compatibility problems, and it may take a considerable amount of time to learn to use the tool. Given the potential for delays, it is recommended that evaluation process is initiated as early as possible.

The evaluation criteria typically is so comprehensive that all of it may not be covered within the time available for evaluation. Therefore, the ranking of importance of evaluating each criteria should be used as a guideline in evaluation. Nevertheless, it is still quite likely that not all data for the criteria is available. Missing data will need to be handled in the analysis process.

The results of the evaluation phase will be documented using the evaluation criteria template defined by the evaluation criteria definition process. There are two particular tasks in the evaluation. The *cost to baseline* estimate will be used in calculating the financial cost figure for the alternative. The qualitative characteristics of each alternative are evaluated thought the "tests", which can be a measurements, experiments or other pieces of information about the alternatives. The results of such tests are documented, together with a qualitative description that elaborates any issues that could be relevant in interpreting the outcome of the test.

The evaluation is completed when all alternatives have been evaluated by the defined criteria or required data has been determined not to be available.

2.5 Analysis of Results

The evaluation of alternatives in the OTSO method concentrates in producing consistent data about the alternatives. We deliberately want to separate the analysis of this data from producing the data. This allows the use of appropriate techniques in evaluation data analysis for decision making.

The analysis process of graphically presented in Figure 4. Note that the hierarchical decomposition of criteria and their weighting has already been done in the criteria definition process. The OTSO method also assumes the use of AHP as a multiple criteria decision making technique.

The decision of using COTS software is based on the estimated costs of an COTS alternative and the estimated value it will bring to the project. The cost and value estimation has two challenges. First, as with all estimation problems, it is difficult to estimate characteristics that are based on events and processes that have not taken place. The second problem is that each COTS alternative may have distinct characteristics that make their comparison rather difficult, even if reliable cost and feature estimates were available. One alternative may have features that others lack so it is difficult to normalize the costs associated with each alternative.

The OTSO method uses an approach that allows a balanced comparison of alternatives

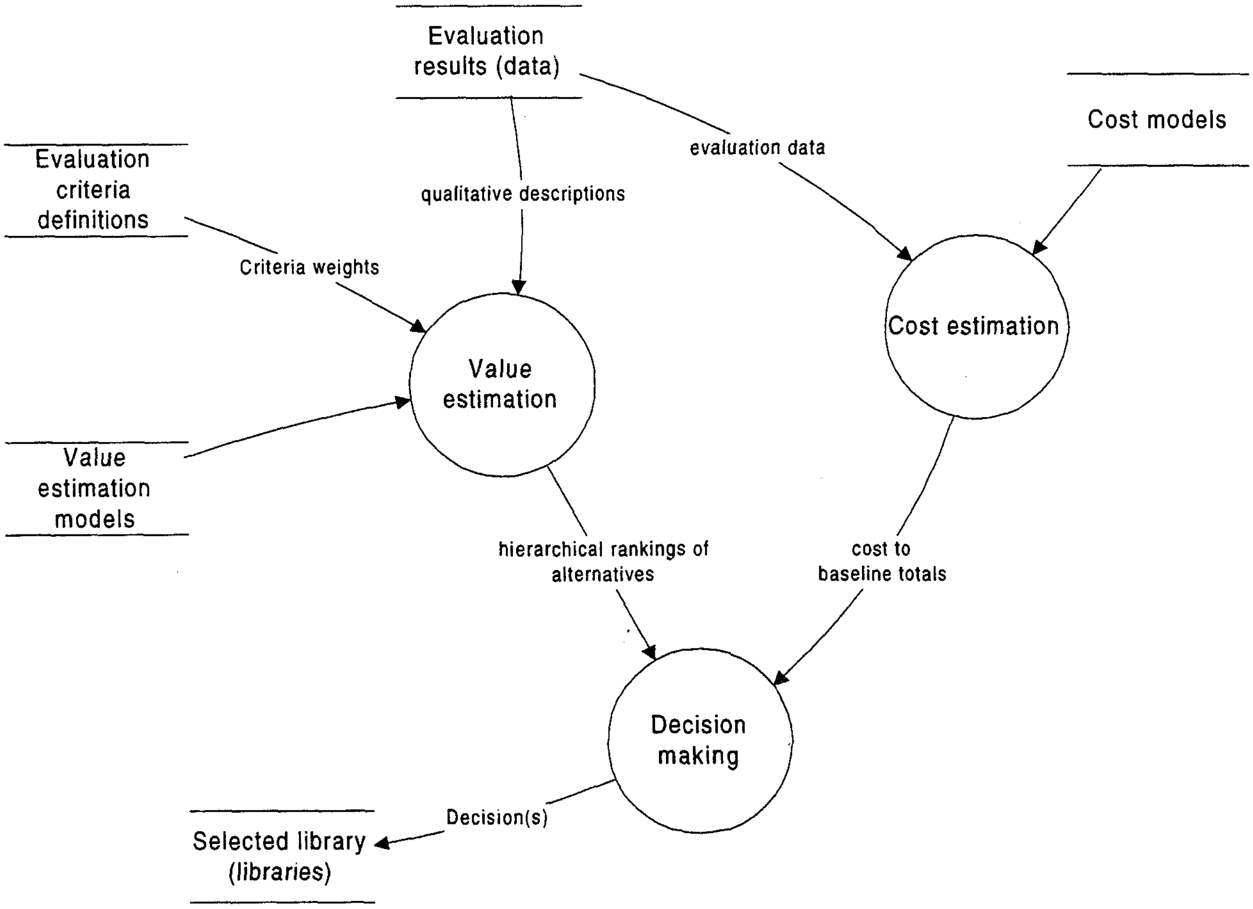


Figure 4: Analysis of results process

with respect to their costs and value they provide. This section presents the general principle of the OTSO cost and value assessment, as well as the individual approach possible for evaluating cost and value separately.

The OTSO cost and value assessment is based on the idea of making all alternatives through a common reference point. This reference point is called the *baseline.* Baseline should be defined as a set of characteristics that each COTS alternative must meet, or exceed, after they have been modified and developed further for the project's purposes. Baseline should reflect realistically the true situation in the project. It should represent the characteristics and features that must be satisfied, no more, no less. The baseline should be derived from the requirement specification and good understanding of the possible implied requirements. The cost estimation for each alternative is based on how much it costs to obtain, develop them further and integrate them to meet the baseline.

The value estimation of each alternative is based on their characteristics assuming that they have been developed further and integrated to meet the baseline. This way, the cost estimation problem is dealt with separately and results in cost estimates that are comparable with respect to the baseline requirements. The value estimation is based on the baseline reference point and each alternative is rewarded for characteristics that exceed the baseline.

The idea behind the baseline as a basis for cost and value estimation can be illustrated with the help of Figure 5 [17]. The different characteristics that are relevant for the baseline are presented on x axis. The Figure 5 assumes that each characteristic can be expressed as a vector, in terms of being able to refer to how well it meets the baseline. Each vector represents how well each alternative (A_i) satisfies the characteristics (c_j) defined in the baseline. A vector can be referenced by defining the alternative and the characteristic in question (A_ic_j). Examples of possible characteristics include:

- stated functionality (e.g., "zoom in capability" or "supports background printing")
- quality characteristic (e.g., "reliability", "level of documentation")
- consumption of resources (e.g., "disk space required")
- standards compliance (e.g., "Win95 compatible", "matches our coding standards")

Figure 5 shows an example situation where an alternative's situation is presented as a set of vectors. Baseline is defined as a set of vectors B_j and represented by a horizontal, zagged bold line in Figure 5. Alternative A_1's current vector set is represented by vectors A_1c_j, where j=1,...10. The cost estimation problem for A_1 is to estimate the cost of "upgrading" A_1's characteristics to meet the baseline.

2.5.1 Estimating the Cost of COTS software

Two main approaches for cost estimation are available: use of cost models or work breakdown structure analysis. Cost models, in theory, may provide a way to obtain unbiased estimates but their problem is that traditional cost models are not very applicable for COTS software cost estimation and it is difficult to capture all relevant factors into a single cost model. A COTS specific cost model has been recently developed and this may provide a partial solution to such cost estimation [11]. However, this kind of model may require customization and calibration for each organization to be effective.

The work breakdown structure analysis may be, for many organizations, the feasible method for COTS software cost estimation: the development and integration tasks for a COTS software are listed and decomposed, effort for each task estimated and total effort summed up. The disadvantage of this approach is that it can be very sensitive to bias or the experience of the personnel.

The OTSO method does not address what method or model is used for COTS software reuse cost estimation. Whatever approach is used, the OTSO method extends the financial COTS software evaluation by allowing the consideration of other factors that may influence the decision. Examples of such factors include the consideration of features that exceed the requirement specification, quality characteristics that are not included in the cost estimation model (e.g., reliability, maintainability, portability, efficiency, etc.), and business or strategic issues that may influence the decision. These issues can sometimes be decisive in COTS software selection and cost estimation alone cannot effectively cover these aspects.

The costs of acquiring COTS software can be broken down to three main classes: acquisition costs, further development costs and integration costs. The OTSO method contains a template for breaking these down further to support COTS software cost estimation (Table 1). While the acquisition costs are relatively straightforward to estimate, the further development costs and integration costs present much more challenging problems.

The further development costs of COTS products are based on developing them to meet the baseline. However, as the baseline may be difficult to define accurately and as few organizations have accurate COTS software cost estimation models or expertise, this cost estimate may have a large margin of error.

Sometimes COTS software includes features that were not originally required for the application. We refer to such functionality or characteristics features as *unrequired features.* Dealing with these unrequired features may

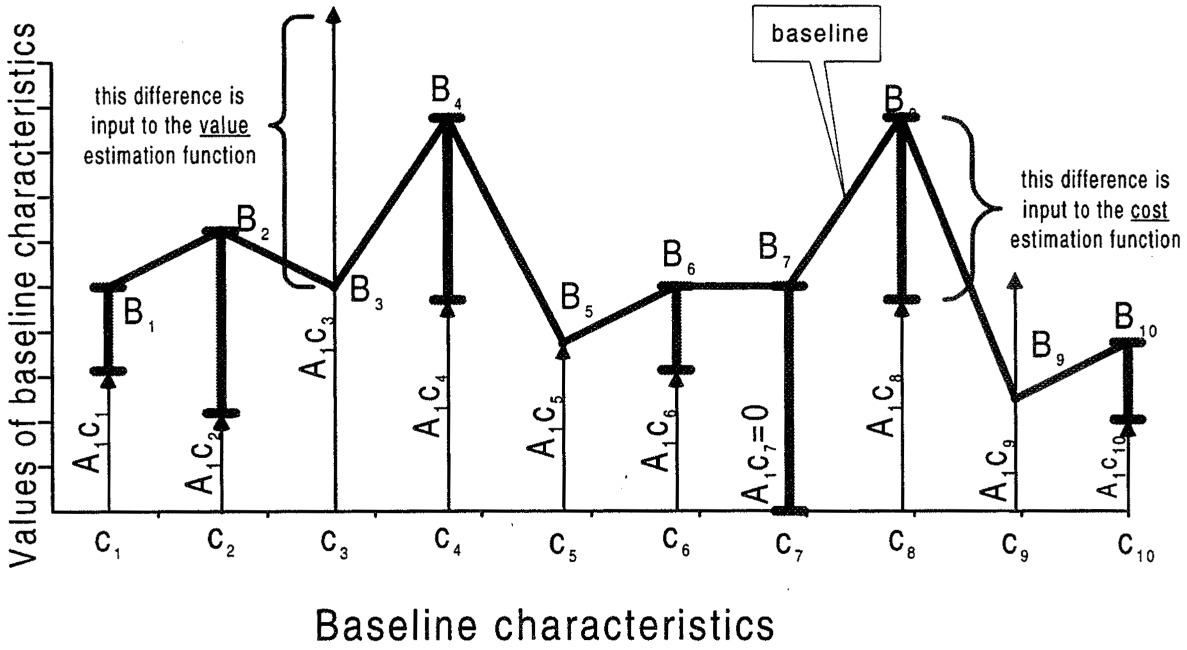


Figure 5: The baseline estimation principle.

Cost item	Explanation
Acquisition costs	
Development version acquisition costs	The cost involved in obtaining an adequate number of licenses for software development
Delivery (run-time) version costs	The possible costs of obtaining the right to deliver the COTS software as a part of the software to users.
Maintenance costs	The possible maintenance fees for the COTS software, e.g., for obtaining the rights for version upgrades or bug patches.
Learning costs	Cost of providing training, independent learning and the impact of learning curve effect on productivity.
Infrastructure upgrade costs	The possible costs involved in having to upgrade some parts of the computer or software environment, e.g., obtaining additional memory capacity, upgrading operating system versions or obtaining necessary support software.
Further development costs	
Cost to develop the COTS software to meet the baseline requirements	The cost of making changes to the COTS software so that it meets the requirements set for the system.
Integration costs	
Modification costs	The cost of making modifications to COTS software.
Costs involved in building additional interface modules	The cost of making interface modules that facilitate the integration of COTS software to the system.
The impact of testing external components	The effort to test COTS software components and correct errors associated with them may be different from the effort to in-house components. However, the COTS software impact can be both positive and negative, depending on the situation.
Increased testing costs for unrequired features	The increased cost of testing the unrequired features that COTS software bring in to the system.

Table 1: Cost components

complicate the cost estimation process. Although some unrequired features may be marginally useful for users, they may make the system too complex for some users. Added functionality may also increase integration and development costs.

We recommend that organizations involved in component reuse initiate procedures to develop and customize their cost models for this purpose. As it takes time to develop these models, most organizations may have to rely on expert judgments in these cost estimates.

2.5.2 Qualitative Analysis of Benefits

As the COTS alternatives have been evaluated the evaluation data needs to be used for making a decision. A common approach for this in many organizations is an approach that can be called the *weighted scoring method* (WSM). The WSM method is typically applied in the following fashion: criteria are defined and each criterion is assigned a weight or a score. In the case of using weights, they may be normalized so that their total is one. If "scoring" is used, this is done, e.g., by assigning a "weight score" between one and five for each criterion. Then, each alternative is given a score on each criterion. The score for each alternative is counted by the following formula:

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

where subscript *a* represents an alternative and *n* represents the number of criteria. There are several shortcomings in this approach and it is questionable whether WSM can represent true preferences between alternatives [17].

The OTSO method relies on the use of the Analytic Hierarchy Process (AHP) for consolidating the evaluation data for decision making purposes. The AHP technique was developed by Thomas Saaty for multiple criteria decision making situations [26,27]. The technique has been widely and successfully used in several fields [28], including software engineering [12] and software selection [15,21]. It has been reported to be an effective technique in multiple criteria decision making situations in several case studies and experiments [10,13,28,31]. Due to the hierarchical treatment of our criteria, AHP fits well into our evaluation process as well. AHP is supported by a commercial tool that supports the entering of judgments and performs all the necessary calculations [29].

The AHP is based on the idea of decomposing a multiple criteria decision making problem into a hierarchy of criteria. At each level in the hierarchy the relative importance of factors is assessed by pair-wise comparisons. Finally, the alternatives are compared in pairs with respect to the criteria.

From our perspective the main advantage of AHP is that it provides a systematic, validated approach for consolidating information about alternatives using multiple criteria. AHP can be used to "add up" the characteristics of each alternative. Furthermore, an additional benefit of AHP is that we can choose the level of consolidation. We recommend that consolidation is only carried out to the level that is possible without sacrificing important information. On the other hand, some consolidation may be necessary in order not to overflow the decision makers with too much detailed, unstructured information.

3. Case Studies

We have carried out two case studies using the OTSO method. The first case study was aimed at assessing the overall feasibility of the method and the second one focused on the comparison of analysis methods. Both case studies took place in the NASA/EOS program and were dealing with real software development projects facing a COTS selection problem.

Our first case study, the ReMap project [17], dealt with the selection of a library that would be used to develop an interactive, geographical user interface for entering location information on Earth's surface areas. This case study used the OTSO method's hierarchical and detailed criteria definition approach. The cost to baseline principle was applied separately after the only feasible alternative was selected.

In addition to providing feedback to the development of the OTSO method, the ReMap project lead us to make some other observations that are useful in similar selection processes. First, it was necessary to refine the stated requirements significantly in order to develop a meaningful evaluation criteria set. We believe that this is a common phenomenon. When the COTS software selection process takes place, requirements are typically not defined in much detail. Yet detailed requirement definitions are necessary for evaluating different products. The interaction of the COTS software selection process and requirements definition process is essential. We also witnessed that evaluating COTS alternatives not only helped in *refining* the requirements, it also lead to *extending* of requirements in some situations. This is an additional challenge in requirements management. In our case the extended requirements were limited to the area of the COTS software but it is also quite conceivable that the evaluation process influences the whole system requirements.

Second, a considerable amount of calendar time may need to be spent on installation and logistics before the evaluation can commence. In our case, this limited the time available for detailed evaluation. If the COTS software selection is in the critical path in the project, some attention needs to be placed on the logistics and procurement so that unnecessary delays are avoided. Overall, it seems that the actual effort spent on evaluating each alternative was actually not very high but the calendar time elapsed was.

Finally, despite all the efforts in criteria definition and evaluation, there will inevitably some data that is missing. This may be because the data is simply not available, because it would be too costly to obtain the data or the data is not available in time. The main conclusion of the ReMap case study was that the OTSO method seemed to be a feasible approach in COTS software selection and its overhead costs are rather marginal.

The second case study dealt with the selection of a hypertext browser for the EOS information service [1,17-20]. The objectives of the second case study [18] were to (i) validate the feasibility of the evaluation criteria definition approach in the OTSO method and (ii) compare the AHP and the weighted scoring method for analyzing the data. Our hypothesis was that the more detailed evaluation criteria definition will result in more effective, consistent and reliable evaluation process. We also expected that the AHP method would give decision makers more confidence in the decisions they made.

The details about the selection process were collected and this case study included a comparison between two analysis methods, the AHP method and a weighted scoring method. This case study further supported our conclusion of the low overhead of the OTSO method. Furthermore, the second case study involved several evaluators and our criteria definition approach improved the efficiency and consistency of the evaluation. The second case study also had an unexpected result when the two analysis methods were compared: they yielded different results, i.e., the rankings of the COTS alternatives were different with the two analysis methods, even though they were based on the same data.

4. Conclusions

We have presented the main characteristics of the OTSO method for COTS software selection. The method addresses an important software development activity that, to our knowledge, has not been addressed by the reuse research community extensively. The method supports systematic evaluation of COTS alternatives and considers both financial and qualitative aspects of the selection process.

The experiences from our case studies indicate that the method is feasible in operational context, it improves the efficiency and consistency of evaluations, it has low overhead costs, and it makes the COTS software selection decision rationale explicit in the organization. The detailed evaluation criteria also contribute to the refinement of application requirements. We also observed that the selection process can be very sensitive to the method used in analyzing the evaluation data.

Although the initial experiences from the method are encouraging, further and more formal experiments are required to validate the method. At the moment we are particularly concerned about the method's sensitivity to the cost estimation method used, as it is a strong factor in the decision making process. Also, the feasibility of documenting the baseline in adequate detail may be an issue that limits the applicability of our method in larger applications.

5. References

[1] J. D. Baker. Planet Earth, The View from Space, H. Friedman (Ed). Cambridge, Massachusetts: Harvard University Press, 1990.

[2] B. Barnes, T. Durek, J. Gaffney, and A. Pyster. A Framework and Economic Foundation for Software Reuse. In: Tutorial: Software Reuse: Emerging Technology, ed. W. Tracz. Washington: IEEE Computer Society, 1988.pp. 77-88.

[3] V. R. Basili, Software Modeling and Measurement: The Goal/Question/Metric Paradigm CS-TR-2956, 1992. Computer Science Technical Report Series. University of Maryland. College Park, MD.

[4] V. R. Basili, G. Caldiera, and G. Cantone, A Reference Architecture for the Component Factory, ACM Transactions on Software Engineering and Methodology, vol. 1, 1. pp. 53-80, 1992.

[5] 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.

[6] V. R. Basili and H. D. Rombach, Tailoring the Software Process to Project Goals and Environments, pp. 345-357, 1987. Proceedings of the 9th International Conference on Software Engineering. IEEE Computer Society Press.

[7] T. Birgerstaff and C. Richter, Reusability Framework, Assessment, and Directions, IEEE Software, vol. March. pp. 41-49, 1987.

[8] T. B. Bollinger and S. L. Pfleeger, Economics of Reuse: issues and alternatives, Information and Software Technology, vol. 32, 10. pp. 643-652, 1991.

[9] G. Boloix and P. N. Robillard, A Software System Evaluation Framework, IEEE Computer, vol. 28, 12. pp. 17-26, 1995.

[10] A. T. W. Chu and R. E. Kalaba, A Comparison of Two Methods for Determining the Weights Belonging to Fuzzy Sets, Journal of Optimization Theory and Applications, vol. 27, 4. pp. 531-538, 1979.

[11] T. Ellis, COTS Integration in Software Solutions - A Cost Model, 1995. Proceedings of the NCOSE International Symposium "Systems Engineering in the Global Marketplace".

[12] G. R. Finnie, G. E. Wittig, and D. I. Petkov, Prioritizing Software Development Productivity Factors Using the Analytic Hierarchy Process, Journal of Systems and Software, vol. 22, pp. 129-139, 1995.

[13] E. H. Forman, Facts and Fictions about the Analytic Hierarchy Process, Mathematical and Computer Modelling, vol. 17, 4-5. pp. 19-26, 1993.

[14] M. L. Griss, Software reuse: From library to factory, IBM Systems Journal, vol. 32, 4. pp. 548-566, 1993.

[15] S. Hong and R. Nigam. Analytic Hierarchy Process Applied to Evaluation of Financial Modeling Software. In: Proceedings of the 1st International Conference on Decision Support Systems, Atlanta, GA, Anonymous1981.

[16] J. W. Hooper and R. O. Chester. Software Reuse: Guidelines and Methods, R.A. Demillo (Ed). New York: Plenum Press, 1991.

[17] J. Kontio, OTSO: A Systematic Process for Reusable Software Component Selection CS-TR-3478, 1995. University of Maryland Technical Reports. University of Maryland. College Park, MD.

[18] J. Kontio, A Case Study in Applying a Systematic Method for COTS Selection, 1996. Proceedings of the 18th International Conference on Software Engineering.

[19] J. Kontio and S. Chen, Hypertext Document Viewing Tool Trade Study: Summary of Evaluation Results 441-TP-002-001, 1995. EOS project Technical Paper. Hughes Corporation, EOS project.

[20] J. Kontio, S. Chen, K. Limperos, and J. Hung, Hypertext Document Viewing Tool

Trade Study: Evaluation Criteria Definitions 1995. Internal project documentation. University of Maryland.

[21] H. Min, Selection of Software: The Analytic Hierarchy Process, International Journal of Physical Distribution & Logistics Management, vol. 22, 1. pp. 42-52, 1992.

[22] S. L. Pfleeger and T. B. Bollinger, The Economics of Reuse: New Approaches to Modeling and Assessing Cost, Information and Software Technology, vol. 1994.

[23] J. S. Poulin, J. M. Caruso, and D. R. Hancock, The business case for software reuse, IBM Systems Journal, vol. 32, 4. pp. 567-594, 1993.

[24] R. Prieto-Díaz, Implementing faceted classification for software reuse, Communications of the ACM, vol. 34, 5.1991.

[25] C. V. Ramamoorthy, V. Garg, and A. Prakash, Support for Reusability in Genesis, pp. 299-305, 1986. Proceedings of Compsac 86. Chicago.

[26] T. L. Saaty. Decision Making for Leaders, Belmont, California: Lifetime Learning Publications, 1982. 291 pages.

[27] T. L. Saaty. The Analytic Hierarchy Process, New York: McGraw-Hill, 1990. 287 pages.

[28] T. L. Saaty. Analytic Hierarchy. In: Encyclopedia of Science & Technology, Anonymous McGraw-Hill, 1992.pp. 559-563.

[29] T. L. Saaty, Expert Choice software 1995, ver. 9, rel. 1995. Expert Choice Inc. IBM. DOS.

[30] W. Schäfer, R. Prieto-Díaz, and M. Matsumoto. Software Reusability, W. Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds). Hemel Hempstead: Ellis Horwood, 1994.

[31] P. J. Schoemaker and C. C. Waid, An Experimental Comparison of Different Approaches to Determining Weights in Additive Utility Models, Management Science, vol. 28, 2. pp. 182-196, 1982.

[32] W. Tracz, Reusability Comes of Age, IEEE Software, vol. July. pp. 6-8, 1987.

[33] W. Tracz. Software Reuse: Motivators and Inhibitors. In: Tutorial: Software Reuse: Emerging Technology, ed. W. Tracz. Washington: IEEE Computer Society, 1988.pp. 62-67.

[34] W. Tracz. Tutorial: Software Reuse: Emerging Technology, W. Tracz (Ed). Washington: IEEE Computer Society, 1988.

[35] W. Tracz, Legal obligations for software reuse, American Programmer, vol. March.1991.

[36] M. Wasmund, Implementing Critical Success Factors in software reuse, IBM Systems Journal, vol. 32, 4. pp. 595-611, 1993.

[37] F. Wolff, Long-term controlling of software reuse, Information and Software Technology, vol. 34, 3. pp. 178-184, 1992.

A COTS Selection Method and Experiences of Its Use

Jyrki Kontio
Roseanne Tesoriero
Gianluigi Caldiera
University of Maryland
Computer Science Department
A.V.Williams Building
College Park, MD 20742, U.S.A.
[jkontio, roseanne, gcaldiera]@cs.umd.edu

Show-Fune Chen
Kevin Limperos
Mike Deutsch
Hughes Information Technology Corporation
1616A McCormick Dr.
Landover, MD 20785-5372, U.S.A.
[schen, klimpero, miked]@eos.hitc.com

The EOS Program

- ECS (EOSDIS Core System) is designed to collect, process, store, and distribute Earth Science related data to the Earth Science Community.
- The ECS Flight Operations System is designed to control ECS spacecraft and deliver raw data to the Ground System.
- The ECS Ground System is being developed as a series of subsystems, described below, each with its own piece of the overall ECS design.

Introduction

- Reuse is increasing
  - systems are developed from components
  - pressure to reuse from customers or corporate policies
- External sources for Reuse
  - more alternatives for reuse: standard platforms and OS, more competition, better access to products
- Many projects have the option to select from several reuse candidates
- COTS = Commercial Off-The-Shelf Software, this presentation also applies to other reusable components (commercial, public domain, internal, etc..)

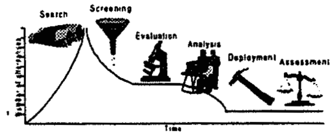
The Problem

- Reuse decisions are frequent in large projects
  - ➨ a lot time time and effort spent, high potential impact on the final product
- Reuse decision process is rarely well defined and formalized
  - ➨ each project reinvents the wheel under schedule pressure
- Selection criteria often over emphasize technical criteria
  - ➨ user requirements receive less attention
- Simplistic techniques are often used for consolidating evaluation results
  - ➨ may bias conclusions
- COTS alternatives may be difficult to compare
  - ➨ "Apples and oranges" -- some provide additional value, cost structure may be different

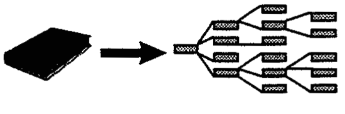
➨Reusable component selection process needs to be supported

The OTSO Method

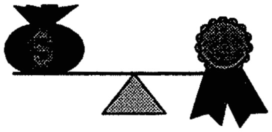
- A defined COTS selection process



- Requirements-driven, explicit and detailed evaluation criteria definition



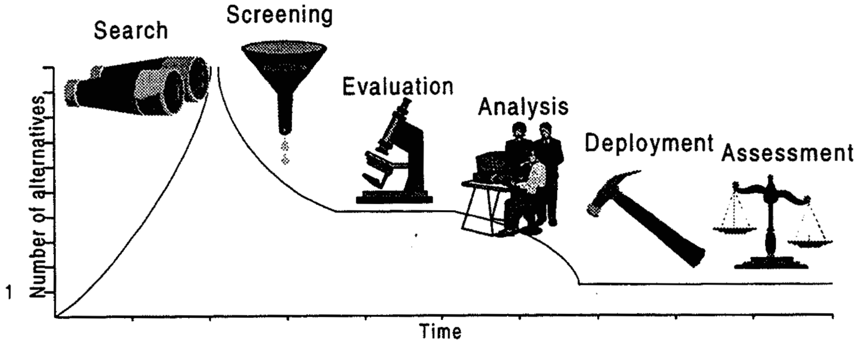
- A model for comparing the cost and value of different COTS alternatives



- Use of a reliable method for synthesizing evaluation results

COTS Selection Process

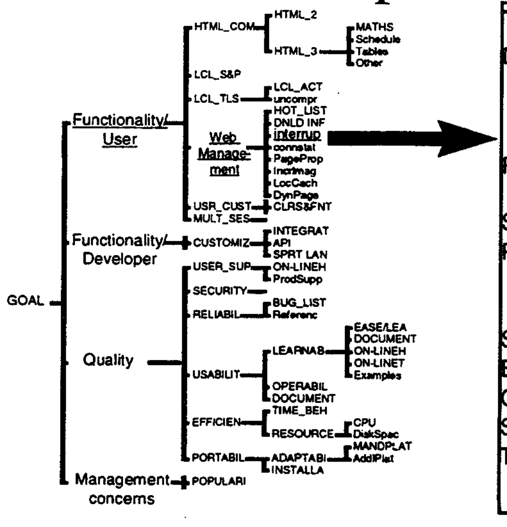
- The selection phases have different goals
- Each phase can be defined and supported



Evaluation Criteria

- Hierarchical decomposition of criteria <u>from requirements</u>
  - in practice, this involves refinement of requirements, e.g:
    - functional requirements: required functionality of the COTS
    - quality: e.g., maintainability, reliability, portability, etc.
    - management/strategic concerns: will the vendor be there in the future, what is its market share, future development plans
- Definition of "tests" for each criteria
  - an observation, test, or a metric that can be used to characterize a criterion
- "Operational definitions" for all "tests"
  - unambiguous definitions for the criteria and "tests"
  - limits evaluator freedom but produces more consistent results
- Criteria evolve over time
  - the "Search" and "Screening" phases can deal with few general criteria

Evaluation Criteria/Example



Test: Interrupt of retrieval

Definition	Does the tool support interrupting of retrievals and how the interrupts are controlled.
Rationale	Retrievals may need to be canceled occasionally.
Scale	Free format description
Possible values	no interrupt support interrupt all retrievals selective interrupt of retrievals
Screening rule	no
Baseline	NA
Qualitative desc.	
Source	Tests done with the tools.
Test priority	Recommended

Cost Estimation Approaches

- Cost models
  - using COTS specific cost models to estimate costs, e.g., Cocomo or function point variations
  - Requires calibration and historical data
  - Accumulates experience but may ignore situation specific issues

- Cost component breakdown
  - Different cost components are identified and estimated

Cost item
Acquisition costs
Development version acquisitio costs
Delivery (run-time) version costs
Maintenance costs
Learning costs
Infrastructure upgrade costs
Costs to baseline
Cost to develop the (C)OTS to meet the baseline requirements
Integration costs
Modification costs
Costs involved in building additional interface modules
The impact of testing external components
Increased testing costs for unrequired features

Value Estimation

- Development cost < > value
  - especially when you have other options than developing it all yourself

- Multiple criteria approach
  - several factors affect the value (i.e., utility)
  - alternatives can be ranked more reliably by using multiple criteria decision support tools
  - Recommended technique: **Analytic Hierarchy Process (AHP)** by Saaty
    - sound theoretical basis
    - strong empirical validation
    - widely used
    - tool support

Cost + Value Estimation

1. Define the **baseline**: the minimum set of characteristics that the COTS must have
   - include work that you would really have to do if you selected the alternative
2. Assess the **cost to baseline** by cost estimation techniques
   - use the cost estimation method that is suitable for your organization
3. Assume that each alternative will be develop to meet the baseline, **assess the value** of alternatives
   - use the AHP approach
   - define criteria, evaluate alternatives and elicit preferences
   - tool support is a practical necessity
4. Present the results to decision makers
   - Consolidate information according to their preferences

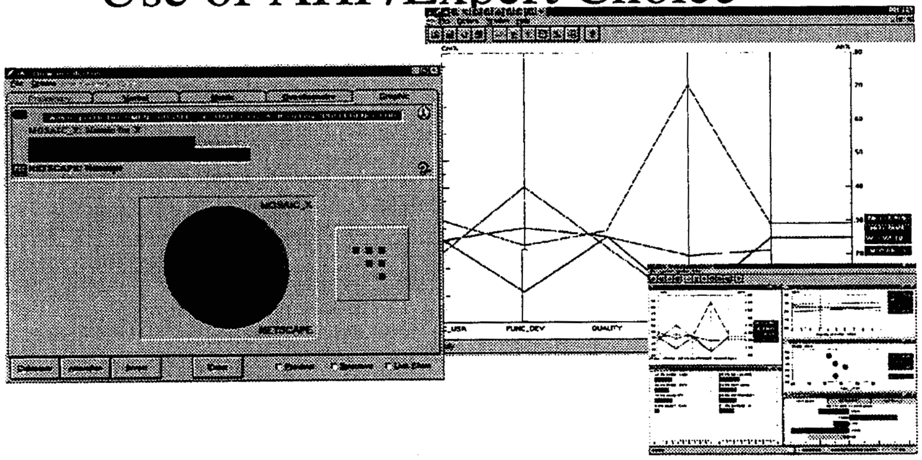
Consolidating the Information

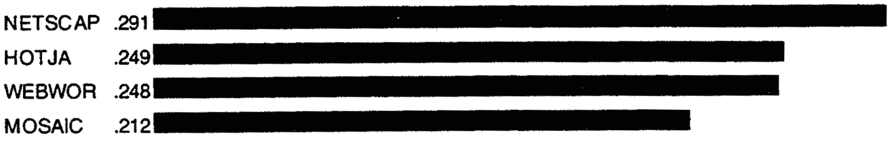
- Weighted scoring method (WSM) common
  - "define criteria ⇨ define weights ⇨ define scores ⇨ multiply and add ⇨ pick the highest score"
- AHP is a widely used decision support technique, supported by a tool (Expert Choice)
- The AHP process:
  - Define criteria (hierarchical decomposition)
  - Weigh criteria by pair-wise comparisons
  - Rank alternatives by pair-wise comparisons
- AHP benefits:
  - Systematic and sound elicitation of judgments
  - Ratio scale preferences between alternatives
  - Results can be presented in a way that decision makers understand the rationale

Consolidating the Information/ WSM Example

Criteria/tests	weight	Mosaic	Netscape	Webworks	HotJava
Test: Level 2 compatibility	**5**	3	3	3	3
Test: HTML Level 3 compatibility schedul	**5**	0	3	0	0
Test: Support for tables	**5**	0	5	0	0
Test: Display of mathematical equations ε	**3**	0	0		2
Test: Other HTML Level 3 features currer	**3**		3		
Test: Local save and print tests	**5**	5	5	5	2
Test: Local tool activation	**5**	5	5	5	5
[illegible]					
Test: Installation problem list	**3**	5	5	5	5
Test: Popularity of the tool	**4**	3	5	0	0
Score		**470.0**	**591.0**	**467.0**	**427.0**
rankings ——►		**2**	**1**	**3**	**4**

Use of AHP/Expert Choice





Browser Selection Effort

Activity	Effort (hrs)		%
Search		20	14%
Screening		8	6%
Evaluation		79	55%
Criteria definition	40		28%
Mosaic for X	10		7%
Netscape	9		6%
Webworks	9.5		7%
HotJava	10.5		7%
Analysis/WSM		5	3%
Analysis/AHP		7	5%
Administration (planning meetings, reporting, etc.)		20	14%
Learning about the methods		1	1%
other (vendor contacts, installations)		4	3%
Total		**144**	

Summary

- OTSO is a reusable selection process that supports organizational learning
- Formalization and support of COTS selection process does not require a large effort, yet it seems to result in more efficient selection process
- Detailed and requirements-based evaluation criteria lead to consistent and understandable decisions
- Cost and value of COTS are different things
- The method of consolidating evaluation results matters, AHP seems to be more reliable than the weighted scoring method

Main References

- J. Kontio, OTSO: A Systematic Process for Reusable Software Component Selection CS-TR-3478, UMIACS-TR-95-63, 1995. University of Maryland Technical Reports. University of Maryland. College Park, MD.
- J. Kontio and S. Chen, Hypertext Document Viewing Tool Trade Study: Summary of Evaluation Results 441-TP-002-001, 1995. EOS project Technical Paper. Hughes Corporation, EOS project.
- T. L. Saaty. *The Analytic Hierarchy Process*, New York: McGraw-Hill, 1990.
- T. L. Saaty. Analytic Hierarchy. In: *Encyclopedia of Science & Technology*, McGraw-Hill, 1992.
- Expert Choice software 1995, ver. 9.0, rel. 1995. Expert Choice Inc. Windows 95.

More information:

- ◆ email: jkontio@cs.umd.edu
- ◆ WWW: http://www.cs.umd.edu/projects/SoftEng/tame/

Process Enactment Within An Environment

Roseanne Tesoriero and Marvin Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

Environment research has often centered on either the set of tools needed to support software development or on the set of process steps followed by personnel on a project as they complete their activities. In this paper, we address the effects that the environment has on the development process in order to complete a project. In particular, we are interested in how software process steps are actually performed using a typical programming environment. We then introduce a model to measure a software engineering process in order to be able to determine the relative tradeoffs among manual process steps and automated environmental tools. Understanding process complexity is a potential result of this model. Data from the Flight Dynamics Division at NASA Goddard Space Flight Center is used to understand these issues.

1 Introduction

Since 1976 the Software engineering Laboratory[1] (SEL) at the NASA Goddard Space Flight Center (GSFC) has been studying flight dynamics software development and has produced over 300 papers and reports describing numerous process improvement technologies. Figure 1 briefly summarizes this activity and outlines the SEL approach to understand, assess, and package technologies useful for the flight dynamics domain. Many of these technologies (e.g., resource models, cleanroom, IV&V, object-oriented design) have been studied and some made part of the SEL development process.

There are two interesting issues not answerable by the set of studies addressed by this figure:

1. Software productivity has obviously improved in the 20 years since the SEL started (and 20 years worth of workshop proceedings attest to that). But by how much? Also, *everyone's* productivity has improved in these past 20 years. Is the SEL doing relatively better than other development organizations? How do we understand this and measure this improvement?
2. What impact has technology (i.e., tools) had on this improvement? Most SEL studies are on *processes* that personnel undertake in the develoment of software. What impact has the set of computers, workstations, and software had on development productivity and quality?

The first issue can be explained by the following analogy. Assume an organization purchased four PCs over the past 15 years:

- 8088 4.77Mhz PC-XT with 10 Mbyte disk bought in 1983 for $5,300
- 80286 6Mhz PC-AT with 20 Mbyte disk bought in 1984 for $4,000
- I486Dx2 66Mhz PC with 340 Mbyte disk bought in 1993 for $2,600
- Pentium 75Mhz PC with 850 Mbyte disk bought in 1995 for $1,700

The fourth system (the Pentium) is obviously the fastest most powerful system of the four. However, the more interesting question is which is most powerful relative to the era in which they were available? Is this organization doing better or worse than other organizations in its industry relative to computer power if it followed this purchase plan?

For answering this question, we have many hardware performance measures: LINPACK, TPC-A,

[1] A joint activity of NASA/GSFC Flight Dynamics Division, Computer Sciences Corporation (CSC), and the University of Maryland.

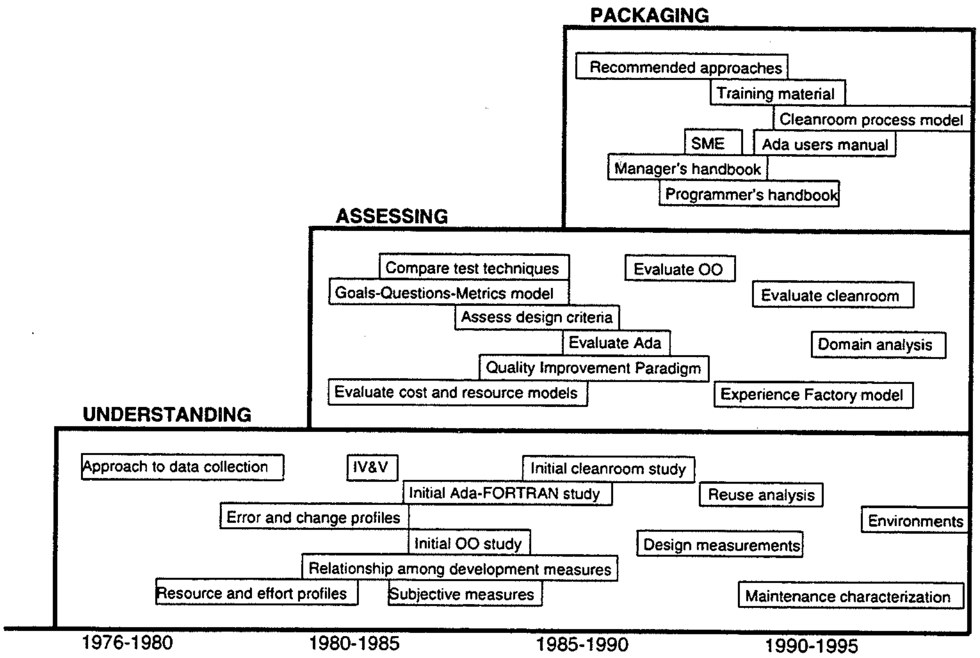


Figure 1: Summary of SEL activities.

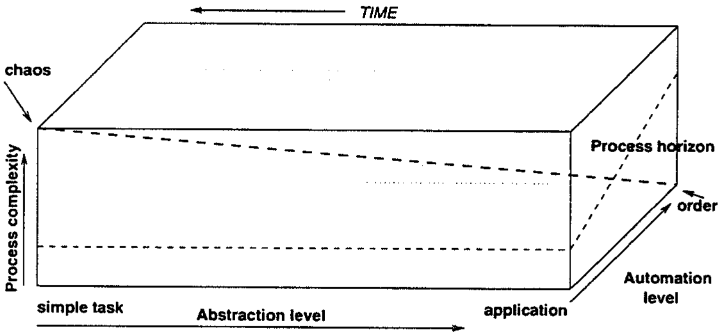


Figure 2: ITEM Model.

$/mips, dhrystones, etc. All have their faults, but at least they generate a set of numbers that can be discussed. How would we do the same for software development practices? We have no such measures of performance. It is toward this end we are trying to develop a model.

This then focuses the remainder of this paper. We are interested in a process measure that addresses the following issues:

1. *How do we model such a process performance measure?* We will base our model on a service-based model of an environment.
2. *What is an environmental service?* We will describe such a model of an environment service.
3. *What tools are used in the SEL environment?* We looked at the evolution of tool use within the SEL to see how tool use reflects the set of services in our environment model.
4. *How do we classify flight dynamics processes?* Can we adapt this service-based model to the flight dynamics application domain better? Can we measure the complexity of a process according to this model?

The next four sections of this paper, in turn, addresses each of these questions.

2 How do we model such a process performance measure?

The model we choose to use is an extension to a model developed by Zelkowitz and Cuthill at the National Institute of Standards and Technology. Called the Information Technology Engineering and Measurement (ITEM) model [16], it addresses the role of both automation and process in an environment.

The model (Figure 2) locates a given development activity according to three parameters:
1. Percent automation (ranging from a manual pro-

cess to totally automated one).
2. Service abstraction level (from simple low-level infrastructure to an entire application domain).
3. Process complexity

We are interested in understanding activities from these three perspectives. We want to know how much of a solution the activity addresses. A tool that does compilation is relatively simple, while one that integrates design, compilation, and testing (assuming such a tool existed) would be more complex, and finally one that by pushing a button configures an entire ground support system for NASA would be even more complex. We call this aspect of an environment the *abstraction level* and is represented by its horizontal placement in the figure.

For any specific activity, there are multiple ways to address its solution. Take for example, sorting a list of names. A totally manual process would be to sort each name by hand from a set of cards, each containing one name. A more automated solution would be the UNIX *sort* command on a file of names. This is the *automation level* of that activity.

Finally, for each activity, there is a limit to the complexity that can be attempted in its solution. For example, sorting a list of names where one first translates by hand each name into binary, sorts that binary list by hand, and then translates the binary back into ASCII, probably represents a solution that is fraught with errors. We call such a solution too complex, or above the *process horizon* for that activity. The process horizon is represented by the dotted line in Figure 2. Note that the line rises as the automation level increases – more automated tasks can handle more complex solutions due to the increased use of computer-based tools that handle the mundane part of the solution.

The point on the model labeled "chaos" represents an extremely low-level manual task of great complexity. This represents a chaotic state of development. On the other hand, the point labeled "order" represents a totally automated solution of the application domain with simple complexity. This is our desired goal. So our problem is to try and describe the location of each process in this ITEM structure and then to show that over time we progress toward the point labeled "order."

Time is a parameter that makes the application of this model even more complex. As technology improves over time, complex tasks of last year are now relatively simple. A complex task like assembly code in 1955 was replaced by the similarly complex task of compiling a Pascal program in 1970, which was replaced by the more similarly complex task of writing a C++ or Ada program today. We can quibble with the term "similiarly complex," but the basic concept is that all of these tasks represented about the same level of effort in their respective eras. However, today, the same assembly language compiler would be considered a more complex lower-level activity than the Ada compiler, and thus would be to the left and higher (more process complexity) in the ITEM figure. This concept is called *technological drift.* All technology moves to the left over time in the ITEM model. A future goal (one we would like to solve, but dare not even try yet) is to measure this flow of technology over time.

3 What is an environmental service?

The ITEM model describes an abstraction level as a way to characterize the functionality of a given activity. For the software development application domain, two service-based models have been developed that address this functional – the NIST/ECMA framework for software engineering environments and the Project Support Environment (PSE) reference model.

3.1 Reference Models

In 1989, the European Computer Manufacturers Association (ECMA) produced a model for the description of services useful for supporting software development activities. In 1991, the National Institute of Standards and Technology (NIST) joined with ECMA to develop the current Edition 3 of this model, known as the NIST/ECMA software engineering environment frameworks reference model [11].

The NIST/ECMA model, also known as the "toaster model" based on a graphic that was developed by George Tatge of HP (Figure 3), defines the underlying infrastructure set of services for supporting tools executing on a software engineering environment. The model consists of 66 services catalogued according to the classification of Figure 3 plus a seventh Operating System set of services that supports the other six categories. Software products, called tools, are added to the environment and interact among one another and with the environment itself by using the operations defined within the seven classes of infrastructure services.

While the NIST/ECMA model defines the under-

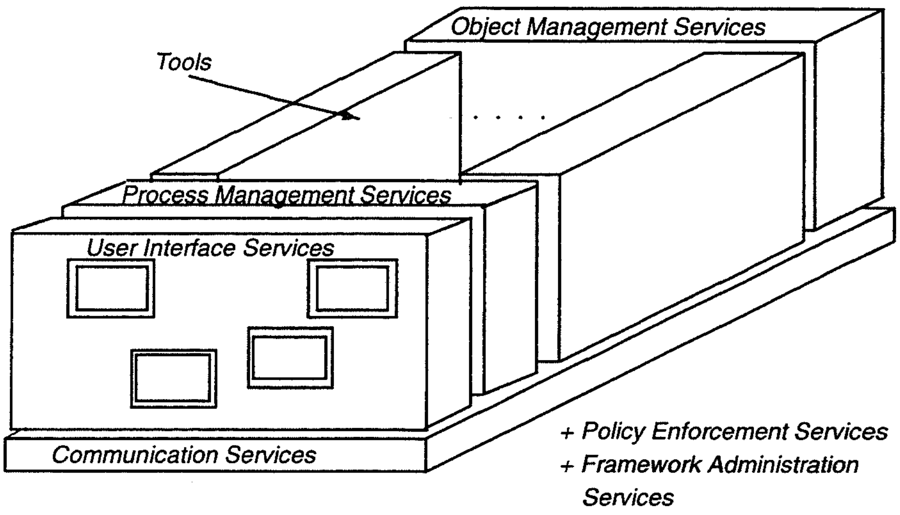

Object Management Services
Tools
. . . .
Process Management Services
User Interface Services
Communication Services
+ Policy Enforcement Services
+ Framework Administration Services

Figure 3: Framework Reference Model Service Groupings.

lying infrastructure for supporting tools within an environment, there was also a need to define the set of tools that users would need for application development within the environment. In order to address this functionality, the U.S. Navy's Next Generation Computer Resources (NGCR) program created the Project Support Environment Standards Working Group (PSESWG). PSESWG developed the Project Support Environment (PSE) Reference Model of the set of services needed to support users of software engineering environments [5]. This model included the NIST/ECMA framework model as the core set of framework services, but adds a structure on the "tool slots" of the framework model.

Services in the PSE model are either *end-user* or *framework*. The former services directly support the execution of a project (i.e., services that tend to be used by those who directly participate in the execution of a project such as programmers, managers, and secretaries). The latter services generally pertain to the operation of the computer system itself (e.g., a human user performing such activities as tool installation) or are used directly by other services in the environment. End-user services are further subdivided into Technical Engineering, Technical Management, Project Management, and Support service categories.

For example, within the Software Engineering group of services within the Technical Engineering category are the common software development activities, such as software design, Compilation, Debugging, etc. Project management services include project scheduling, estimation, and tracking. Support services include activities such as editing, publishing, electronic mail, and other supporting activities.

The model has been used to map (i.e., describe and contrast) the functionality of various products or standards in order to determine how the functionality they provide compares to the functionality present in the model [14].

3.2 Service Hierarchy

Process Steps: We will define a process step as the smallest identified self-contained activity performed in the development of a software project. Personnel on a project perform process steps, and the underlying computer environment executes one or more *tasks* in response to those steps. For many process steps, there is a one-to-one correspondence between process steps and tasks (e.g., editing a file and a tool such as VI or EMACS, placing a file under configuration management and tools such as PANVALET or SCCS). In other instances the process step is clearly defined by a small set of tasks (e.g., creating a software module requires the tasks of editing and compiling). However, other process steps may require manual components as part of the process step, which may not involve the use of a computer (and its execution of tasks) (e.g., software design, inspections, software quality assurance). In these cases, process steps may be implemented by a sequence of actions using process execution notation, such as MARVEL [10].

Reference model services: The tasks provided by an environment can be mapped into the end-user services defined in the reference model. These may also be viewed as an initial set of process steps useful in the software development process.

The services in the reference model represent a broad range of capabilities. Some are clearly crucial to the success of a project (e.g., the existence of a compiler for the compilation process step). Without these services, the development of software would be impossible.

Other services, while not necessities, help to improve the quality and reliability of a software system. For example, program verification helps to improve the correctness and reliability of a program. However, the difficulty of using verification tools and the general lack of availability of effective verification tools means that most development processes omit a verification step.

One way of representing the relative complexity of the services is to impose a hierarchy on them. The levels in the hierarchy indicate increasing levels of complexity in the services provided by the project support environment. Because of advancements in technology and research, we believe that the levels will continuously evolve and change. Currently, the services of the lower levels of the hierarchy represent steps in the software process that are well defined and understood. We can easily identify tools that provide these services. As the levels of the hierarchy increase, the current understanding of the process steps required to provide the services of the level decreases. It becomes increasingly difficult to find examples of tools which provide these services. We believe that over time, as the knowledge and understanding of complex services become clearer, the services will drop to the lower levels of the service hierarchy.

Process Step	Sample tasks (tools)
Compilation	Ada, C, or FORTRAN compilers, preprocessors
Debugging	symbolic debuggers
Software Build	make
Text Processing	editors such as VI or EMACS

Table 1: Level 1 Services.

We propose the following four complexity levels of the reference model end-user services:

Level 1: Basic Services. These represent the crucial processes in software development. Software development would be impossible without them, and all are present in every realistic development. In addition, they are all implemented as single tasks in an environment. The process steps and example tools for these services are given in Table 1.

Process Step	Sample tasks (tools)
Configuration Mgmt.	PANVALET, CMS, SCCS
Software Generation*	LEX, YACC, 4GLs, IDL tools
Software Testing	Software TestWorks
Software Static Analysis*	source code analyzers, SAP
Numeric Processing	spreadsheets such as QuattroPro and 1-2-3
Figure Processing	MacDraw, MacPaint, xfig
Mail	ccMail, Eudora
Bulletin Board*	Readnews, ccMail
Tool Installation and Customization	UNIX .rc tool customization files
Host-Target Connection	Xterm, ftp, telnet
Audio and Video Processing*	MBone
Calendar and Reminder*	Synchronize, CRON
Conferencing*	Unix talk
Information Mgmt.*	Archie, Gopher, yellow pages, parts lists, Xmosaic, Netscape
Planning	TimeLine, MacProject
Estimation	Software Management Environment (SME), tools that implement cost models
Tracking	SME, Gantt and Pert charts)
Data Interchange*	binhex, uuencode/uudecode, Kermit
Publishing	Tex, Latex, Framemaker, PowerPoint, MS Word

* Optional process steps

Table 2: Level 2 Services.

Level 2: Single Task Services. These services represent the process steps that are usually performed in an environment. The process steps often exist as single tasks in an environment and tools to automate them are readily available. Most of the listed services are typically utilized in an environment; however, some, although automated and readily available, are not necessarily required. Those services that are considered to be optional are indicated by an * in Table 2.

Process Step	Sample tasks (tools)
System Integration	
System Testing	
System Static Analysis	
Software Requirements Engineering	OOATool and DCDS
Software Design	IDE's Software through Pictures (StP), Teamwork ObjectMaker
Software Simulation and Modeling	Simula, screen simulators
Software Traceability	ORCA (Object-based Requirements Capture and Analysis) and RETRAC (REquirements TRACeability)
Change Management	Netherworld, ChangeVision
Reuse Management	Asset Source for Software Engineering Technology (ASSET), Central Archive for Reusable Defense Software (CARDS)
Metrics	Amadeus, COCOMO (COst COntainment MOdel), cyclomatic complexity structure models
Annotation	
Process Management	MARVEL, shell scripts
Target Monitoring	

Table 3: Level 3 Services.

Level 3: Composite Services. The services at this level represent process steps that are required in software development, but there is rarely a single tool which can be used to accomplish all of the activities of the process step. These process steps generally represent the state of the art in software engineering research. As the table in Table 3 demonstrates, there are not as many tools available for for implementing these as single tasks within an environment. As technology evolves and improves, it is expected that these services will become single task services.

Level 4: Advanced Services. The services in this level represent needed research in software engineering. Not every project incorporates these process steps into their development process. Often, if tools are available for these services, they are experimental. It is expected that as more research and experimentation is completed, these services will move to Level 3 services. The process steps and examples for these services are given in Table 4.

Process Step	Sample tasks (tools)
System Requirements Engineering	
System Design and Allocation	configuration languages and systems (Polylith, Rapide)
System Simulation and Modeling	Performance Oriented Design (POD) and Synthetic Environments for Requirements and Concepts Evaluation and Synthesis (SERCES)
System Traceability	ORCA (Object-based Requirements Capture and Analysis), RETRAC (REquirements TRACeability)
Software Reverse Engineering	
Software Re-engineering	
Process Definition	state transition charts, MARVEL, action diagrams, Petri nets
Process Library	Process Asset Library (PAL)
Process Exchange	
Process Usage	EAST, Cohesion
Risk Analysis	
System Re-engineering	
Software Verification	Cleanroom, Z, VDM
Policy Enforcement	

Table 4: Level 4 Services.

4 Tools are used in the SEL environment

In order to study tool use and process interaction, we are using data collected by the NASA/GSFC SEL. The SEL has been collecting data since 1976 on over 125 ground support software projects for unmanned spacecraft developed by the Flight Dynamics Division. The data in this paper is a result of reading history reports on lessons learned from approximately 20 such recent projects, and from interviews with personnel who built this software.

4.1 Flight Dynamics Software Development

Two classes of products form the core of the work within this division:

(1) Attitude Ground Support Software (AGSS)

provides attitude determination capabilities for determining spacecraft location and orientation (e.g., where is it and in which direction is it pointing at a given time in the future). This is needed to coordinate spacecraft location with the data collected by on-board scientific instruments and for spacecraft orbit modification.

(2) Simulators for testing onboard computer capabilities before launch.

Until the late 1980s, all source programs were written in FORTRAN. AGSS software is still written in FORTRAN, while most simulators are now written in Ada. Other software is written in FORTRAN, in Ada, or in C. There is now a project investigating the applicability of C++ in this environment.

The programming (e.g., hardware and software) environment at NASA has generally passed through three distinct phases since the SEL was created in 1976. Initially, all work was performed on IBM-compatible mainframes. In the early 1980s, the DEC VAX computer was used for some development, while the mainframes were still the operational computers for spacecraft control. In the late 1980s, PCs started to appear on desktops and were sometimes used for initial editing and compilation of modules. Today there are some initial moves to use UNIX workstations and build systems using a client-server architecture. The following briefly describes this evolution.

IBM Mainframe Environment

The mainframe environment is the oldest of the three development environments. It is also the least sophisticated. The mainframe environment is used for the development of AGSS systems with FORTRAN being the programming language used for development. In order to integrate the various tools available in the IBM mainframe environment, the Software Development Environment (SDE) was built. SDE is an application that was built on top of ISPF (Interactive System Productivity Facility). It is a menu-driven facility that provides a common interface to the tools available in this environment (Table 5).

Activity	**Tool**
Design	DesignAid CASE 2000, MacDraw, CorelDraw
Code	SDE*, ISPF, QED , VS FORTRAN, assembler, linker, RXVP80, ICA, GESS Display Builder*
Configuration Mgmt.	PANVALET
Test	CAT

* – Tool developed for use within this environment

Table 5: Tools in the IBM mainframe environment.

Design: In the mainframe environment, there is limited automated support for software design. On three projects (GRO, UARS and EUVE), DesignAid CASE 2000 was used for design. The history reports indicate that the tool was useful in some areas such as drawing design diagrams and organizing documentation; however, it was not utilized to its fullest potential. One reason cited is that the task of re-entering data flow diagrams and data dictionaries from the functional specifications document to the PC was too time consuming.

The reuse rates for these early FORTRAN projects were low, approximately 10 percent, while later FORTRAN projects were able to achieve reuse rates between 80 and 90 percent. Part of the increase in reuse rate may be attributed to the creation of two FORTRAN libraries that handle nearly 80 percent of the functionality of new ground support systems. It is interesting to note that the high reuse rates were achieved without the support of specific reuse tools.

Code: QED (a line editor) and ISPF (a screen editor) are the editors used in this environment. To create executable code, users use the FORTRAN compiler, the IBM assembler and the IBM linkage editor. Other than editors and compilers, there are few other tools available for use. RXVP80 and ICA are tools for static analysis of FORTRAN code used to detect inconsistencies in program structure or misuse of variables. To generate code for displays, the GESS Display Builder, developed and maintained by NASA over the past 20 years, is used. This tool was to be replaced after the COBE project ended in 1988; however, no replacement has yet been developed.

There is no symbolic debugger available to mainframe users. To compensate for this deficiency, developers typically place debugging statements in their code. On the TONS project, Microsoft FORTRAN and CodeView were used to build code on PCs initially. When the code was considered complete, it was recompiled with the mainframe FORTRAN compiler.

Configuration Management: For configuration management, the environment offers PANVALET, a commercial source code management system. PANVALET has been the exclusive source of configuration management in the mainframe environment for over 12 years.

Test: There are no standard tools available for software testing and verification. Data for testing are generated with the assistance of FDD developed simulators or by small programs developed by testers. Any software problems that are found while executing the test plan are reported by filling out a Software Trouble Report (STR). On one project, SAMPEX, the STR form was automated. There are no standard tools that support the generation of test plans or the tracking of test cases. The Configuration Analysis Tool (CAT) was used on the ERBS and COBE projects to track discrepancies and test cases. The capabilities of this tool were described as being useful, but limited. Usually, the tracking of the status of software problems is done through spreadsheets on Macintoshes or PCs, as was done on the COBE project.

Although there are no specific tools for test case generation and analysis, the quality of the code has improved over time. The defect rates for early FORTRAN projects were 9.8 development errors per thousand lines of code, while the defect rates for more recent FORTRAN projects are 4.8 development errors per thousand lines of code.

VAX Environment

In 1985 the FDD began to experiment with the use of Ada for software development. It was found that the mainframe Ada was not reliable enough for AGSS development, but the DEC VAX computer provided an environment suitable for simulator development [15]. Although the initial plan was to completely transition from the mainframe environment to an Ada development environment on the VAX, the VAX environment has been used for the development of telemetry and dynamic simulators only. To support Ada development, this environment has the Ada Compilation System (ACS). Users of the ACS have expressed satisfaction with the capabilities provided by this system. Table 6 lists the tools that are available in the VAX environment.

Activity	Tool
Design	System Architect, StP
Code	ACS, EDT, LSE, Ada compiler, assembler, linker, debugger, PCA, SCA
Configuration Mgmt.	CMS
Test	None

Table 6: Tools in the VAX environment.

Design: There are no standard tools used for the design of projects on the VAX, although several projects have experimented with a few CASE tools. System Architect was used on TONS. The diagrams produced by the tool were considered to be poor in quality. The learning curve and the unavailability of multiple copies of the tool discouraged future use of the tool. The WINDPOLR project experimented with Software through Pictures (StP). The project experienced difficulties with consistency and configuration control when they tried to make changes to the design. The use of StP was discontinued in the design phase.

Initial reuse rates for Ada projects range between 5 and 20 percent. After the FDD became more familiar with the Ada language and began to experiment with object-oriented techniques, the reuse rates increased. More recently, the reuse rates for Ada projects are closer to 90 percent for projects with similar domains. As with mainframe projects, the high reuse rates have been accomplished without the use of specific reuse tools. However, as mentioned previously, higher reuse also occurred in the mainframe FORTRAN environment. This implies that increased reuse is not resulting only from the use of the Ada language; the use of object-oriented technology is having an across-the-board effect on productivity.

Code: The ACS provides various services necessary for software development. For editing, a screen editor (EDT) and a language sensitive editor (LSE) are available. To produce executables, the VAX Ada Compiler, VAX assembler, and linker are available. The Performance Code Analyzer (PCA) and the Source Code Analyzer (SCA) provide dynamic and static analysis of code. Configuration management is handled by the Code Management System (CMS). Unlike the mainframe environment, the VAX environment provides a symbolic debugger. Additionally, VAX users can use electronic mail for communication.

For project management, the Software Management Environment (SME) has been used by the WINDPOLR and EUVE projects. SME accesses the SEL database of past project histories and plots the current project's attributes over time (e.g., errors per week, effort per week) as compared to these histories. CSC's Performance Measurement System (PMS) was used to track project resources in terms of schedule, performance and cost by tasks on the GRO and UARS projects. Both of these tools run on PCs.

Test: There are no standard tools used for testing in the VAX environment. The testing process is carried out similar to the way it is done in the mainframe environment.

As with FORTRAN programs, reliability of Ada code has improved over the past 10 years. The defect rates for the early Ada projects were 10.5 development errors per thousand lines of code. More recently, the defect rates are 4.0 development errors per thousand lines of code.

Workstation Environment

Over the past few years, several projects have been developed on PCs and HP workstations running UNIX. Most new development is expected to transition to this environment over the next few years. There have only been a handful of workstation projects so far: a multi-application user interface system (UIX); multi-application attitude support components (GSS R1); and a mission AGSS application (XTE AGSS). For development on PCs, the Santa Cruz Operation (SCO) Open Desktop environment was used. For the workstations, HP's desktop environment was used. Table 7 lists the tools available in the workstation environment.

Activity	Tool
Design	MacDraw, Microsoft Word
Code	SCO Unix, HP Desktop
Configuration Mgmt.	PVCS, SCCS
Test	None

Table 7: Tools in the workstation environment.

Design: No design tools have been used on any of the workstation projects. Microsoft's Word was used to draw design diagrams and create the design documents.

The design process for the workstation projects differed from the traditional mainframe and VAX design process. The conventional design process in the FDD calls for all component designs to be completed before the critical design review. For the XTE AGSS project, the first AGSS built in the open systems workstation environment, the design process was modified to include iteration. Each build of the system was designed then coded successively. This led to difficulties in estimating and assessing the progress of the project. Productivity on the initial builds was considerably lower than expected and the size of the applications was underestimated by a factor of three [4].

The difficulty in transitioning to the workstation environment was greater than anticipated. The perspective of the developers had to change from a purely software engineering one to a systems engineering perspective. Because the infrastructures of the mainframe and VAX environments were stable, the FDD developers had little experience with system engineering techniques. This led to unexpected difficulties in design.

Code: The desktop environments include tools for editing, compilation and debugging. Because they are UNIX environments, the standard UNIX commands (e.g., diff, grep, ftp, make) are also available. To build X/Motif windows, Builder Xcessory was used on the XTE project. There are several tools that are available in this environment, but, they have not been used successfully. For example, SCCS is available for source code management, but, for the XTE project, the UNIX file system was used instead. This implies a more manual error-prone process to make sure that only appropriate versions of a system are used at the appropriate time. Similarly, a performance profiler (probe) was not used successfully during the XTE project.

The use of tools such as Builder Xcessory for interface design also indicates a growing trend towards the use of Commercial Off-the-Shelf (COTS) software components in future FDD systems. In the past, because of a stable computing environment, the FDD tended to build customized software development toolsets and supporting infrastructures. When development moved to the workstation environment, the problems addressed by the customized toolsets and infrastructure had to be resolved again.

An Ada development tool set was considered for the workstation environment. Because of the cost, matching the capabilities available on the VAX was not considered as feasible. So, Ada development is still done on the VAX and integrated and tested on the workstations.

Configuration Management: SCCS was attempted for source code management on the XTE project, but it was not used successfully. The UNIX file system was used instead. Additionally, PVCS was also used on both the PCs and HP workstations for version control.

Test: For testing, the process for workstations is similar to the VAX and mainframe test process. Internally developed simulators generate and send data for testing. The simulators used for the XTE and UIX projects include the TPOCC internal simulator and the GMIS simulator.

Additional Support

History reports discuss tools specific for the development of flight dynamics software. However, many other technologies are standard in this environment and are not so mentioned. No one today considers devices such as telephones, fax machines, or copies as significant technologies in doing business, yet all are crucial to project success.

The same is true of some computer tools. Electronic mail is all pervasive for communicating among project members, although connections to others via the Internet is not particularly efficient. The NASA/GSFC mail system has undergone several changes over the past few years, and according to the authors of this report, is still substandard with communication to the world outside of GSFC being very slow. ccMail is the current mail system of choice.

In all three of the environments, the activities of documentation and management are supported by the same tools. Table 8 indicates the tools used for these activities.

There were several tools that were used to support the workstation projects that were not UNIX tools. For instance, MacDraw and CorelDraw were used to create diagrams. These are general purpose graphics programs and not designed specifically for program design applications. General purpose word processors, such as Microsoft Word and PC-Write were used for word processing. QuattroPro is the spreadsheet that was used for tracking status. Additionally, CSC's software estimation spreadsheet, BEMS/BPMS was used. BPMS/BEMS is a spreadsheet tool that has been developed by CSC for project planning and estimation. Depending upon the life cycle model chosen for the project, the tool generates estimated start and completion dates for all phases of the life cycle. The tool is also used for producing charts and reports with the planning and estimation information.

Activity	Tool
Documentation	Microsoft Word, PC Write
Management	SME*, PMS*, BPMS/BEMS*, QuattroPro

* – Tool developed for use within this environment

Table 8: Tools in the support environment.

4.2 Development Models

In describing the three NASA environments, the discussion centered on the tools that were used (e.g., ISPF, SME, Ada compiler, MS Word) and the processes used to develop modules (e.g., design, testing, documentation). What is the relationship between the processes that need to be accomplished and the set of tools that can be executed to help solve development problems? In this section, we look at the concepts of software processes and programming environments. We describe formal models of each. In the next section, we unify both concepts in order to describe the process architecture of the FDD development environment.

Processes

For our purposes, we use the term *process* to mean a set of partially ordered process steps intended to reach a goal [6]. A process is a fairly complex undertaking, such as software design or configuration management. Performing such processes involves the enactment of many of these process steps.

A *process step* is defined as a subprocess of a process or any recursively defined process step. Creating a module or building a version of the system from the configuration management library are generally considered as process steps. An elementary process step is called an *activity*. Activities are composed of *tasks*, the simpliest action under consideration in a development. For the most part, tasks are single executions of a tool in an environment. Compiling a program, editing a file, or checking in a module are considered to be tasks.

There are various approaches to examining process development. One approach is to form a generalized, *ideal* model and refine it to a specific instantiation. The Software Engineering Institute's (SEI) Capability Maturity Model (CMM) for Software [12] and process modeling languages and environments are often used in this manner. Alternatively, a model can evolve and be extracted from data based on previous experiences. This is the approach taken by the Software Engineering Laboratory's Experience Factory. The Experience Factory builds models based on knowledge gained from past software projects and experiments.

Process-centered Software Engineering Environments. As described in [7], a process-centered software engineering environment provides assistance to its users by interpreting explicit guidance-oriented or enforcement-oriented software process models. The generic process interpretor, the process engine, is the heart of a process-centered software engineering environment. Guidance-oriented process models give the process performer indirect assistance while enforcement-oriented process models give direct assistance. Often, such environments follow scripts, developed by the process engineer, which define the sequence of actions to undertake. Such scripts often look like programming language source programs. For example, to create a new module, a simple script could look something like:

```
Loop
    Call Editor (module name)
    Call Compiler (module name)
    If errors, repeat loop
    Call Testing Program
    If errors, repeat loop
End Loop
Check (module name) in Configuration Library
```

It should seem clear that this script will iteratively call the editor as long as either the compiler or testing program finds errors. Programs are only entered in the configuration library if they pass both tests without errors. If the development team is bound by these process rules, complex sets of interactions can be developed for a development team to follow.

Use of such scripting allows for greater control over the development process. For example, one could prohibit checking a module into the configuration library until the testing program succeeded. Data could be collected by automatically calling a data repository program (e.g., Amadeus) at appropriate steps in the script. Cooperative workflow models can be developed as mail is sent from one developer to another informing the new developer of work needing to be done. For example, after checking in a module into the configuration library, an automatic prompt could inform testing personnel that a new module needs to be incorporated into the latest build of the system.

MARVEL [8] [9] [10] is an example of an enforcement-oriented process-centered software engineering environment. The MARVEL environment uses strategies which are predefined by the process engineer. A strategy consists of an objectbase description, tool descriptions, and/or rules that model the software development process. The objectbase description defines the structure (objects) of the project database. The tool descriptions provide the mapping from the objects defined in the objectbase description to the actual location of the tool implementation. The MARVEL rules follow the style of Hoare's assertions for program verification. When the preconditions of an activity are satisfied, it is scheduled for invocation. After the completion of an activity, the postconditions become true and may satisfy the preconditions of another activity. All activities with satisfied preconditions are then scheduled for invocation.

Capability Maturity Model. The CMM [12] is a cross between a software quality approach and a process engineering approach. The CMM identifies key practices that state the fundamental policies, procedures and activities for key process areas. These process areas are grouped into five levels of maturity: (1) Initial; (2) Repeatable; (3) Defined; (4) Managed; and (5) Optimizing. The CMM provides a framework for the practices and areas that should be addressed by a process model, although it doesn't specify the process model that should be used. The individual organization utilizes the framework to build a process model that encompasses the practices identified in the CMM. As an organization matures, the process model covers more of the key process areas at higher levels. The CMM gives the organization a strategy for the steps to be taken for continuous process improvement.

The Software Engineering Laboratory. The Experience Factory was developed by the NASA/GSFC Software Engineering Laboratory as an empirically-based feedback-oriented model of process improvement.

The SEL collects data both manually and automatically. Manual data includes effort data (e.g., time spent by programmers on a variety of tasks: design, coding, testing), error data (e.g., errors or changes, and the effort to find, design and make those changes), and subjective and objective facts about projects (e.g., start and end completion dates, goals and attributes of project and whether they were met). Automatically collected data includes computer use, program static analysis, and source line and module counts.

Software development in the SEL is based upon the Experience Factory concept. The Experience Factory [2] [3] is an infrastructure aimed at capitalizing and reusing the life cycle experience and products. The Experience Factory is the basis for the process model driving FDD product development. The Experience Factory is a logical and physical organization with activities independent from the ones of the development organization. The purpose of the development orgainzation is to develop and deliver systems. It provides the Experience Factory with product development and environment characteristics, data and a diversity of models (resources, quality, product, process) currently used by the projects in order to deliver their capabilities. The Experience Factory processes this information and provides direct feedback to each project activity, together with goals and models tailored from previous project increments. It also produces, stores and provides upon request baselines, tools, lessons learned, and data; all presented from a more generalized perspective.

The distinguishing characteristic of the Experience Factory is that the organization defines itself as continuously improving because it learns from its own business, not from an external, *ideal* process model. Process improvements are based on the understanding of the relationship between process and product in the specific organization.

4.3 Process Enactment in the FDD

In this section we classify the set of processes undertaken as part of NASA/GSFC FDD software development to understand the impact that the underlying computer system has on this development. We will do this by merging the concepts of processes and environment reference models described in the previous section.

FDD Tool Use

From our survey of the three FDD environments, we collected information on various tools used by many projects over the past 10 years (Figure 4). We can classify those tools according to the tasks (i.e., services) that they implement in the PSE model (Figure 5).

The PSE reference model was built around the set of tasks that can be applied to the software development process. For the most part, each PSE service can be mapped to a specific tool that executes within an environment. For some of these services, the mapping is quite simple. The Compilation service obviously maps to the Ada or FORTRAN compiler in the NASA SEL environment. However, other services of the PSE model map to process steps in the NASA environment. For example, there is no single software testing tool in the FDD. Instead, the software testing service becomes a process step enacted as a sequence of tasks, some manual and some automated. We wish to characterize such processes.

In what follows, we use the classification of services in the PSE reference model as a means to characterize the functionality of tools that may appear within an environment. Our goal is to show the relationship between these services and the set of process steps that define the software development process.

FDD Process Enactment Examples

To understand how tools are used in the FDD environment, it is instructive to examine how processes are enacted in the environment. Services can be provided in various ways. For example, a tool set could be integrated to present the user with the necessary services. Integrated toolsets such as Microsoft's Office (Word, Excel, Powerpoint and Access) and IDE's Software through Pictures are commercial examples of this. In such cases the user interacts with a single interface to ease the transition among the various tools in the colleciton. As mentioned earlier, the SEL developed the SDE interface for the mainframe environment in order to provide this integrated set of tools to the developer.

KEY

● Mainframe projects
○ VAX projects
○ Workstation projects
* Current Environment

		AGSS											Simulator														Workstation		
		Current	ERBSAGSS	COBEAGSS	GROAGSS	GOESAGSS	UARSAGSS	EUVEAGSS	SAMPEX	TOMS-EP	FASTAGSS	TONSGSS	Current	POWITS	WINDPOLAR	GOFOR	GOESIM	UARSTELS	GOADA	EUVETELS	UARSDSIM	EUVEDSIM	SAMPEXTS	TOMSTELS	FASTELS	SWASXTLS	Current	XTE AGSS	UIX
IBM tools	PANVALET	*	●	●	●	●	●	●	●	●	●	●			○				○										
	SFORT		●																										
	CAT		●	●																									
	VS FORTRAN	*		●	●	●	●	●	●	●	●	●		○	○														
	RXVP80			●	●	●				●																			
	GESS (Dispiay Builder)	*		●	●	●	●		●	●					○														
	SDE	*		●	●	●	●		●	●		●			○														
	ASSEMBLER H	*		●			●	●	●	●	●																		
	CONVERT			●																									
	PANEXEC			●																									
	TEXT display package			●																									
	ICA	*		●	●		●	●		●		●			○														
	CCC				●																								
	SuperPDL																												
	ISPF	*	●	●	●	●	●	●	●	●	●	●																	
	QED		●	●	●	●	●	●	●	●	●	●																	
VAX tools	ACS											●	*	○				○	○	○		○	○	○	○	○			
	Ada Compiler											●	*	○			○	○	○	○		○	○	○	○	○			
	CMS											●	*	○			○	○	○	○	○	○	○	○	○	○			
	PCA													○		○		○	○	○	○	○	○		○				
	EDT											●	*	○			○		○	○			○		○	○			
	VAX mail								●			●	*	○			○	○	○	○	○	○		○	○	○			
	VAX debugger								●			●	*	○				○	○	○			○		○	○			
	LSE								●			●	*	○				○	○	○	○	○	○		○	○			
	SCA											●				○			○	○	○	○			○				
	TARTAN Ada Compiler											●																	
	SME (PC client)														○							○							
	StP (UNIX)														○														
HP tools	X/Motif																										*	○	○
	HP full screen editor																											○	○
	vi editor																										*	○	○
	HP C compiler																										*	○	○
	HP FORTRAN compiler																										*	○	○
	HP symbolic debugger (xdb)																										*	○	○
	HP desktop environment																											○	○
	Builder Xcessory																										*	○	
	Xterm and VT200 emulator																										*	○	○
	UNIX commands (diff,make,ftp...)																										*	○	○
	GSFC Code 510 code counter																											○	○
	probe																											○	○
	HP VADS Ada Compiler																										*		
	Softbench																										*		
	Xoftware/32																										*		
	HPTRAN																										*		
	TPOCC internal simulator																										*	○	○
	GMIS simulator																										*	○	○
	GSU simulator																										*		
PC tools	PCTRAN																											○	○
	PVCS	*										●															*	○	○
	CorelDraw																											○	○
	MS Word	*											*														*	○	○
	PC-Write	*											*														*	○	○
	ccMail	*											*														*	○	○
	QuattroPro	*																									*	○	○
	BEMS/BPMS	*											*														*	○	○
	SCO Unix																												○
	SCO Open Desktop																												○
	CODEBASE																												○
	DEMO						●																						
	MS FORTRAN											●																	
	MS CodeView											●																	
	STR-generating program								●																				
	PMS	*			●								*								○								
	System Architect											●			○														
	DesignAid CASE 2000				●		●	●							○														
	RIM			●																									
	GSFC telemail						●																						
	MacDraw	*								●	●	●	*	○	○	○			○				○			○	*	○	○
	MATHCAD												*		○								○				*		

Figure 4: FDD tool use.

	IBM TOOLS														
	PANVALET	SFORT	CAT	VS FORTRAN	RXVP80	GESS Display Builder	SDE	ASSEMBLER II	CONVERT	PANEXEC	TEXT Display Package	ICA	CC (Change and Config Control)	ISPF	QED
System Requirements Engineering															
System Design and Allocation															
System Simulation and Modeling															
System Integration															
System Testing															
System Static Analysis															
System Re-engineering															
Host-Target Connection															
Target Monitoring															
Traceability															
Software Requirements Engineering															
Software Design											●				
Software Simulation and Modeling											●				
Software Verification															
Software Generation		●				●			●						
Compilation				●			●	●							
Software Static Analysis															
Debugging															
Software Testing															
Software Build							●								
Software Reverse Engineering															
Software Re-engineering															
Software Traceability					●							●			
Process Definition															
Process Library															
Process Exchange															
Process Usage															
Configuration Management	●												●		
Change Management			●										●		
Information Management															
Reuse Management															
Metrics			●												
Planning															
Estimation			●												
Risk Analysis															
Tracking			●												
Text Processing														●	●
Numeric Processing															
Figure Processing															
Audio and Video Processing															
Calendar and Reminder															
Annotation															
Publishing															
Mail															
Bulletin Board															
Conferencing															
Tool Installation and Customization															
PSE User and Role Management															
PSE Status Monitoring															
PSE Resource Management															
PSE Diagnostic															
PSE Interchange															
PSE Instruction															
PSE User Access															
Object Management															
Process Management															
Communication															
Operating System															
User Interface											●				
Policy Enforcement															

	VAX TOOLS												
	ACS	Ada Compiler	CMS	PCA	EDT	VAX mail	VAX debugger	LSE	SCA	TARTAN Ada Compiler	SuperPDL	SME	StP
System Requirements Engineering													
System Design and Allocation													
System Simulation and Modeling													
System Integration													
System Testing													
System Static Analysis													
System Re-engineering													
Host-Target Connection													
Target Monitoring													
Traceability													
Software Requirements Engineering													●
Software Design											●		●
Software Simulation and Modeling				●									●
Software Verification													
Software Generation													
Compilation	●	●								●			
Software Static Analysis									●				
Debugging							●						
Software Testing													
Software Build	●												
Software Reverse Engineering													
Software Re-engineering													
Software Traceability													
Process Definition													
Process Library													
Process Exchange													
Process Usage													
Configuration Management			●										
Change Management													
Information Management													
Reuse Management													
Metrics												●	
Planning												●	
Estimation												●	
Risk Analysis													
Tracking													
Text Processing					●			●					
Numeric Processing													
Figure Processing													
Audio and Video Processing													
Calendar and Reminder													
Annotation													
Publishing													
Mail						●							
Bulletin Board													
Conferencing													
Tool Installation and Customization													
PSE User and Role Management													
PSE Status Monitoring													
PSE Resource Management													
PSE Diagnostic													
PSE Interchange													
PSE Instruction													
PSE User Access													
Object Management													
Process Management													
Communication													
Operating System													
User Interface													
Policy Enforcement													

	HP TOOLS																	
	XMotif	HP full screen editor	HP C compiler	HP FORTRAN compiler	HP symbolic debugger	HP desktop	Builder Xcessory	Xterm and VT200 emulator	UNIX	GSFC Code 510 code counter	make	HP VADS Ada Compiler	Softbench	Xoftware/32	HPTRAN	TPOCC internal simulator	GMIS simulator	GSFC simulator
System Requirements Engineering																		
System Design and Allocation																		
System Simulation and Modeling																		
System Integration																		
System Testing																		
System Static Analysis																		
System Re-engineering																		
Host-Target Connection								●										
Target Monitoring																		
Traceability																		
Software Requirements Engineering																		
Software Design																		
Software Simulation and Modeling							●				●					●	●	●
Software Verification																		
Software Generation							●								●			
Compilation			●	●								●						
Software Static Analysis																		
Debugging					●													
Software Testing																●	●	●
Software Build																		
Software Reverse Engineering																		
Software Re-engineering																		
Software Traceability																		
Process Definition																		
Process Library																		
Process Exchange																		
Process Usage																		
Configuration Management																		
Change Management																		
Information Management																		
Reuse Management																		
Metrics																		
Planning																		
Estimation																		
Risk Analysis																		
Tracking																		
Text Processing		●																
Numeric Processing																		
Figure Processing																		
Audio and Video Processing																		
Calendar and Reminder																		
Annotation																		
Publishing																		
Mail																		
Bulletin Board																		
Conferencing																		
Tool Installation and Customization																		
PSE User and Role Management																		
PSE Status Monitoring																		
PSE Resource Management																		
PSE Diagnostic																		
PSE Interchange																		
PSE Instruction																		
PSE User Access																		
Object Management																		
Process Management																		
Communication						●												
Operating System									●									
User Interface	●					●							●					
Policy Enforcement																		

	PC TOOLS																					
	PCTRAN	PVCS	CorelDraw	MS Word	PC Write	ccMail	QuattroPro	BEMS/BPMS	SCO Unix	SCO Open Desktop	CODEBASE	DEMO	MS FORTRAN	MS CodeView	STR generating program	CSC's PMS	System Architect	DesignAid CASE 2000	RIM	GSFC telemail	MacDraw	MATHCAD
System Requirements Engineering																						
System Design and Allocation																						
System Simulation and Modeling																						
System Integration																						
System Testing																						
System Static Analysis																						
System Re-engineering																						
Host-Target Connection	●																					
Target Monitoring																						
Traceability																						
Software Requirements Engineering																						
Software Design																	●	●				
Software Simulation and Modeling												●										
Software Verification																						
Software Generation	●																					
Compilation													●									
Software Static Analysis																						
Debugging														●								
Software Testing																						●
Software Build																						
Software Reverse Engineering																						
Software Re-engineering																						
Software Traceability																						
Process Definition																						
Process Library																						
Process Exchange																						
Process Usage																						
Configuration Management		●																				
Change Management															●							
Information Management																			●			
Reuse Management																						
Metrics								●														
Planning								●								●						
Estimation																●						
Risk Analysis																						
Tracking								●								●						
Text Processing				●	●																●	
Numeric Processing							●	●													●	●
Figure Processing			●	●																	●	
Audio and Video Processing																						
Calendar and Reminder																						
Annotation																						
Publishing																						
Mail						●														●		
Bulletin Board																						
Conferencing																						
Tool Installation and Customization																						
PSE User and Role Management																						
PSE Status Monitoring																						
PSE Resource Management																						
PSE Diagnostic																						
PSE Interchange																						
PSE Instruction																						
PSE User Access																						
Object Management											●								●			
Process Management																						
Communication										●												
Operating System									●													
User Interface										●											●	
Policy Enforcement																						

Figure 5: FDD automated tasks.

Alternatively, the user may have to manually change among different tools to obtain the required services. In what follows, we examine examples of the enactment of several processes in the FDD environment.

New Module Development

The Level 1 Services have been defined to be the most fundamental services of software development, without which software cannot be developed. Consider the process steps for creating new software modules in the mainframe environment. Most of the Level 1 Services are represented in this process. Starting with the module's requirements, the developer must design the module, implement it in FORTRAN, test it, and then release it for configuration control. This involves a complex series of interactions using several tools on the mainframe. This is given by Figure 6. The key for Figure 6 is presented in Figure 7.

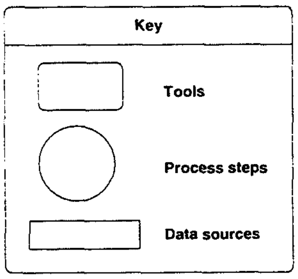


Figure 7: Key for process diagrams.

Several interesting aspects are demonstrated by this figure:

1. No design tool is employed in the Software Design process step. A standard picture-drawing program is used for design documents. This limits the ability to automate traceability back from source programs to specific requirements.

2. The Text Processing process step is employed twice in this process. In one case, an editor (e.g., ISPF) is used to create and modify the modules, while in the other case, a pen is used to complete a paper form for processing the Component Origination Form (COF).

3. There is no debugger available in this environment. To compensate for this, debug statements must be placed in the code and the programmer must run the program to find errors. Notice that debugging is defined by us as a Level 1 Service; however, it is not achieved easily in the mainframe environment.

4. Each activity of the new module development process requires the user to explicitly request the services from each tool. There is no automated sequencing from one activity to the next. For example, there is no guarantee that the design is complete before the Text Processing activity begins or that no errors are found in testing before module is released for Configuration Management. This is not to imply that the developers "cut corners," only that there is no automated process for ensuring that quality control aspects of the process are followed.

As we stated earlier, the FDD is now achieving much higher reuse rates in their software development. How is that achieved? Later we give the Software Reuse process, and indicate how it differs from new module development.

Software Testing

The FDD projects have good reliability rates, yet there are no standard tools available for testing. Typically, previously developed simulators are used to generate and send data to test the new systems.

One explanation for the high reliability in FDD systems may be familiarity. Since the systems constructed by the FDD are very similar in functionality, new systems benefit from the testing done on previous systems. In some cases, up to 90 percent [1] of the functionality is supplied by an existing system. The FDD has become very good at developing the types of systems where they have experience.

When a system on a completely different architecture was attempted, the results were different. There were many problems with the workstation projects [4]. The developers were not prepared for the changes required for a completely different architecture than the ones of previous systems.

Reuse Management

When looking at reuse, two different approaches can be identified:

- Reuse achieved by adjusting the components of an existing system to fit the requirements of a new system.

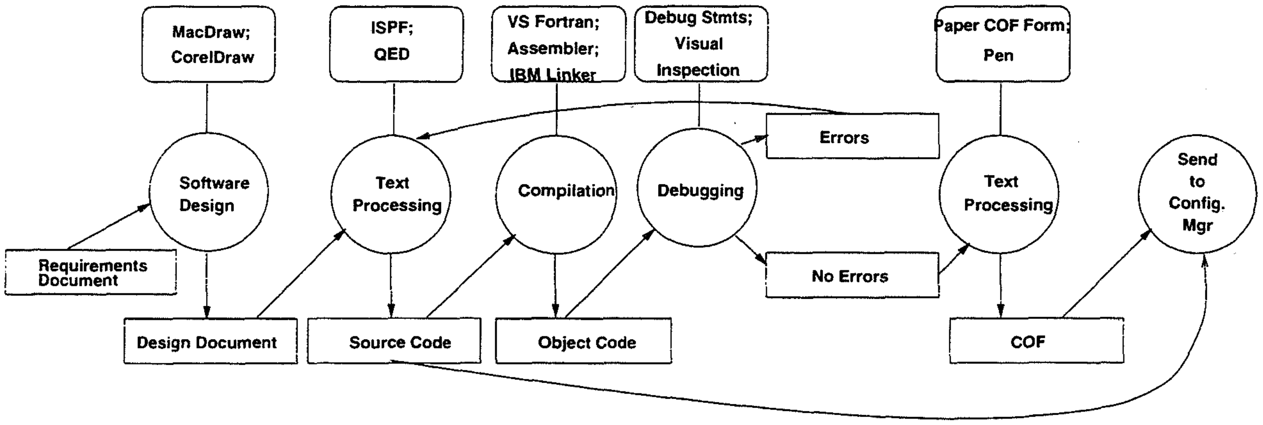


Figure 6: New Module Development Subprocess in the Mainframe Environment.

- Reuse achieved by taking components from various sources and piecing them together to form a new system.

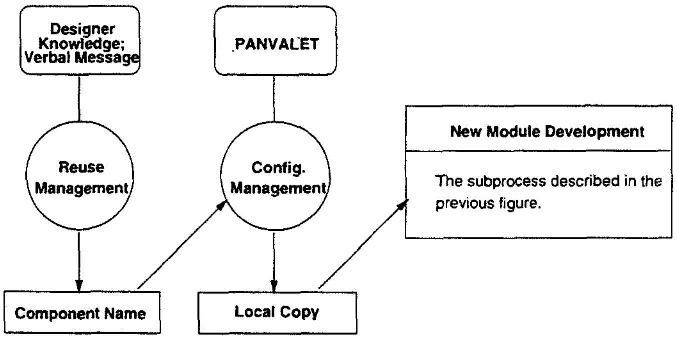


Figure 8: Module Development from a Reusable Component in the Mainframe Environment.

The process for module reuse in the mainframe environment is presented in Figure 8. The most notable feature of this figure is the Reuse Management step of the process. In the FDD, reuse is highly dependent upon the personal knowledge of the designers. Component reuse is achieved by the designer verbally communicating the name of the component to the programmer. Although this does not appear to be an effective approach to reuse management, the FDD has extremely high reuse rates. There are several possible explanations for these results. The FDD tends to use the first of the two approaches mentioned for reuse. The functionality provided by new systems in the FDD are very similar to the functionality of existing systems. Therefore, the code from the components of an existing system can be used verbatim or with slight modifications in a new system. This type of system construction does not require a sophisticated reuse system or process. A designer is not likely to forget about a component from one project to the next. This approach does, however, require a configuration management tool. That could explain why, the existing project support environments of the FDD do not include any reuse tools or reuse mechanisms, but do have several configuration management tools. It also explains the extremely high reuse rates of 80 to 90 percent [1] for the projects.

On the POWITS project, the reuse rate was not as high as previous projects; it was 40 percent [1]. The dip in the reuse rate is attributed to a change in the domain. The simulation software had to be changed from a three-axis stabilized spacecraft to a spin-stabilized spacecraft. To accommodate the change, new software had to be written.

5 Classifying FDD application functions

The final step in this development is to apply a measurement framework to our ITEM model. This workis only preliminary and addresses how we intend to continue this activity.

Our ITEM model depents upon three parameters: abstraction level, automation level, and process complexity. The following is a brief overview of how to impose a metric on top of these concepts.

As shown in Figure 6, a process is simply a graph, where the circles represent the services of the PSE model. As given in Section 3.2, we can classify the complexity of each service. We will define the absstraction level as the set of services from the PSE model that are addressed by the activity. A previously-developed graph complexity model [13] based upon an information theoretic view of program graphs, we can impose a measurement on each such activity. This represents process complexity. For automation level we have the degree of reuse of a given project since this represents the amount of effort that is not being undertaken on a given project. We can then correlate the point in the ITEM model defined by these three numbers with the number of errors or effort expended in developing a given product.

To complete this validation, we need to develop activity graphs for all SEL development processes and then for each project in our database, we need to plot its location on the model. While this will not provide an absolute scale for this model, it should provide a relative evaluation of different developments.

6 Conclusions

After studying this environment, we can separate our conclusions into those that reflect on software development within the FDD at GSFC and on conclusions reflecting process and environment research in general.

6.1 FDD Observations

1. *The overriding concern in this environment is the process for software development and not in the tool support for the developers and managers.* There is considerable care in developing models of how personnel interact. Process-centric methods for development, such as cleanroom and object-oriented technology, have been studied in depth.
2. *When the need for the automation of an activity arises, the FDD tends to develop a tool or search internally for a solution.* The use of SDE in the mainframe environment to achieve a level of tool integration is an example of this. Another example is the use of BPMS/BEMS spreadsheets for project status tracking and estimation. This approach led to difficulties in the transition from the VAX and IBM mainframe environments to an open systems workstation environment.
3. *The environments of the FDD have remained relatively constant over the years.* The FDD does not experiment very often with new tools in their environment. PANVALET has been used for configuration management in the mainframe environment for over 12 years. When a new tool is introduced, it is not given a "fair" chance. For example, StP was only tried one time. The users did not understand the methodology behind the tool. Instead of providing more training, the tool was not used again. This approach is very different from the way in which process-centric methods (e.g. Cleanroom) have been introduced over the years.
4. *Many of the process steps are enacted manually.* For example, there are numerous forms (Component Origination Form, Software Trouble Report) that are required by the SEL to track the software process. Currently, these forms are filled out with pen and paper.
5. *The testing process in the FDD does not utilize conventional software testing tools.* Instead, simulators designed and implemented by the FDD are utilized to generate data for testing. The rest of the testing process is mostly unsupported by tools.
6. *The software design process in the FDD doesn't use conventional design tools.* General purpose drawing tools are used to create design diagrams. This may lead to difficulties in software traceability.

6.2 General Observations

1. *We still need to impose our meaurement framework on top of the ITEM model in order to determine the effectiveness of our measurement approach.* This work is ongoing and will be reported on soon.
2. *The PSE reference model seems to be lacking a level of services.* Although the PSE reference model was useful for examining the coverage of the tools in the environment, it seemed to be missing a level of services. The reference model includes the concepts of infrastructure services as well as complex services (e.g. Software Design). There seems to be an intermediate level of services missing from the model. What are the process steps required for a complex process like software design? The current version of the reference model does not address this issue.
3. *The impact of reuse management tools is unclear.* FDD has high reuse rates, but utilizes no reuse management tools. What is the real impact of reuse management tools? We believe that we have identified two classes of reuse:

(a) Create a system by reusing components from a similar system.

(b) Create a system by using components from various systems or libraries of components.

When discussing reuse, both concepts are usually merged, but they represent very different concepts. In the FDD case, as described earlier, reuse occurs by beginning with the reusable components of an existing system. Perhaps, reuse management tools are only necessary in development projects that assemble components from various sources.

4. *The impact of conventional testing tools is unclear.* FDD has very low error rates, yet uses no specific tools for testing. What is the real impact of testing tools?

References

[1] Bailey, J., Waligora, S., Stark, M., Impact of Ada in the Flight Dynamics Division: excitement and frustration. In Procedings of the 18^{th} Annual Software Engineering Workshop, (Dec, 1993), 422-438.

[2] Basili, V.R., Caldiera, G. and Rombach, H.D., The experience factory. *Encyclopedia of Software Engineering,* Wiley, 1994.

[3] Basili, V.R., Caldiera, G., McGarry, F., Pajerski, R., Page, G. and Waligora S., The Software Engineering Laboratory – an operational software experience factory, ACM/IEEE 14^{th} International Conf. on Soft. Eng., Melbourne, Australia (May, 1992), 370-381.

[4] Boland, D.E., Green, D.S., Steger, W.L., Lessons learned in transitioning to an open systems environment, 19^{th} NASA Software Engineering Workshop, Greenbelt, MD (December 1994), 191-202.

[5] Brown A., Carney D., Oberndorf P. and Zelkowitz M. (Eds), The Project Support Reference Model, Version 2.0, National Institute of Standards and Technology, Special Publication SP 500-213, (November, 1993) (Also CMU/SEI TR 93-TR-23, November, 1993).

[6] Feiler, P.H., Humphrey, W.S., Software process development and enactment: concepts and definitions. In Proceedings of the 2^{nd} Int. Conf. on Software Process, Berlin, Germany (March, 1993), 28-40 (Also CMU/SEI-92-TR-4, 1992).

[7] Lonchamp, J., A structured conceptual and terminological framework for software process engineering. In Proceedings of the 2^{nd} International Conference on the Software Process, Berlin (February, 1993), 41-53.

[8] Kaiser, G.E., Rule-based modeling of the software development process. In Proceedings of the $4th$ International Software Process Workshop, Devon, UK (May, 1988), 84-86.

[9] Kaiser, G.E., A bi-level language for software process modeling. ACM/IEEE 15^{th} International Conf. on Soft. Eng., Baltimore, MD (May, 1993), 132-142.

[10] Kaiser, G.E., Feiler P.H. and Popovich S.S., Intelligent assistance for software development and maintenance. IEEE Software 5(3):40-49, May, 1988.

[11] NIST, Reference Model for Frameworks of Software Engineering Environments Special Publication 500-211, Natl. Inst. of Stnds and Tech., (August, 1993) (Also ECMA TR/55, Edition 3, (June, 1993)).

[12] Paulk, M.C., Curtis, B., Chrissis M.B. and Weber C.V., The Capability Maturity Model for Software, Version 1.1, CMU/SEI-93-TR-24, (1993).

[13] Tian J., and M. V. Zelkowitz, A formal model of program complexity and its application, *J. of Systems and Software* 17, 3 (1992) 253-266.

[14] Zelkowitz M. V., Use of an environment classification model, ACM/IEEE 15^{th} International Conf. on Soft. Eng., Baltimore, MD (May 1993), 348-357.

[15] Zelkowitz M. V., Software engineering technology transfer: Understanding the process, 18^{th} NASA Software Engineering Workshop, Greenbelt, MD (December 1993), 450-458.

[16] Zelkowitz M. V. and B. Cuthill, Application of an information technology model to software engineering environments, *Journal of Sys and Software,* 1996, *(To appear).*

1995 Software Engineering Workshop

Process Enactment Within An Environment

Roseanne Tesoriero and Marvin V. Zelkowitz

Institute for Advanced Computer Studies and

Department of Computer Science

University of Maryland

College Park, Maryland

Background

- 20 years of studies in flight dynamics application domain
  - Most SEL studies were on the effects of processes on the development of products

- SEL obviously is more productive than it was 20 years ago
  - How much has productivity improved since 1976?
  - *But everyone's productivity has improved since 1976.*
  - How much better is productivity now relative to other environments?

- Question: Can we measure process complexity and then apply it to SEL flight dynamics domain?
  - This is the initial phase of an ongoing research effort that grew out of an earlier study on NASA technology transfer efforts. (*See paper in 1993 Software Engineering Workshop proceedings.*)

Process Complexity

- We understand (somewhat) hardware complexity measures
- Which is a more complex system?
  - 8088 4.77Mhz PC-XT with 10 Mbyte disk bought in 1983 for $5,300
  - 80286 6Mhz PC-AT with 20 Mbyte disk bought in 1984 for $4,000
  - I486Dx2 66Mhz PC with 340 Mbyte disk bought in 1993 for $2,600
  - Pentium 75Mhz PC with 850 Mbyte disk bought in 1995 for $1,700
- Pentium is obviously more powerful, but which is more powerful relative to the era in which it was produced?
- We have benchmarks for hardware ($/MIPS, Dhrystone, TPC-A, LINPACK, ...), but can we do the same for software?

Environment Benchmark

- Want a performance measure that addresses effects of processes and tools on productivity
- Measure should be: *Services-produced* per *tool-use* per *life-cycle-activity*
- Rest of talk concerns investigation of this measure:
  - 1. How do we model such a process performance measure?
  - 2. What is an environmental service?
  - 3. What tools are used in the SEL environment?
  - 4. How do we classify flight dynamics application functions?

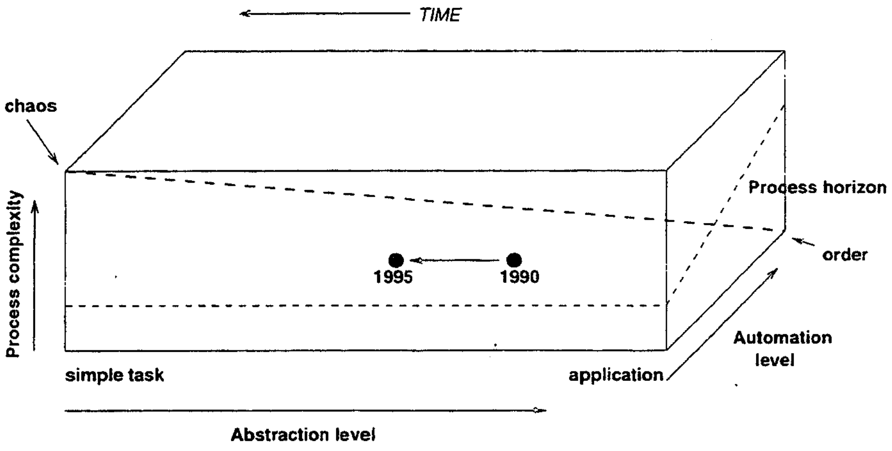
1. How do we model a process measure?

- Assumptions:
  - Increased automation allows for increased complexity
  - There is a limit to process complexity
  - Over time, this complexity increases, due to innovation – *Technological drift* improves all environments each year.

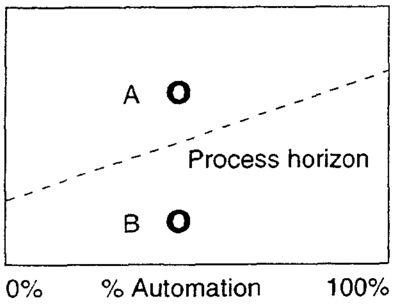
- Use a model based upon work done previously at NIST (ITEM – Information Technology Engineering and Measurement Model)

- Three parameters of model:
  - Percent automation (from manual to totally automated)
  - Service abstraction level (from low-level infrastructure to entire application domain)
  - Process complexity

Overview of Model



Process Horizon

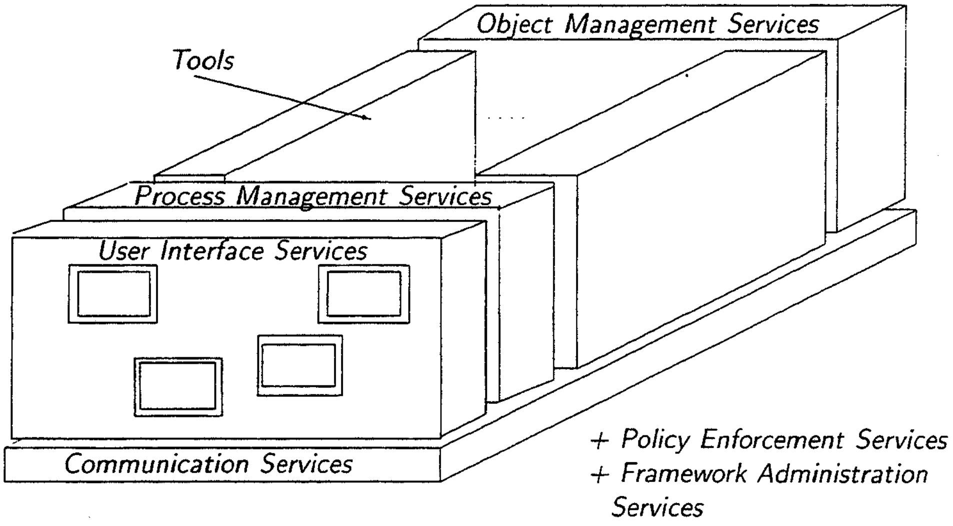


- Increased automation allows for increased complexity
- There is a limit to process complexity
- Need to keep a given process under this horizon at all times
- Over time, this complexity increases, due to innovation

2. What is an environmental function?

- Need a standard reference architecture for describing information flow through an environment
- Reference architecture is a service-based model of the functionality performed by the tools in an environment
- Several models of environments have been proposed
- Two used for this study:
  - NIST – ECMA Software Engineering Environment Framework – Called “toaster model” due to graphic that has been used to describe it
  - Project Support Environment (PSE) reference model of end user services

NIST – ECMA Model


Object Management Services
Tools
Process Management Services
User Interface Services
Communication Services
+ Policy Enforcement Services
+ Framework Administration Services

NIST – ECMA Reference Model

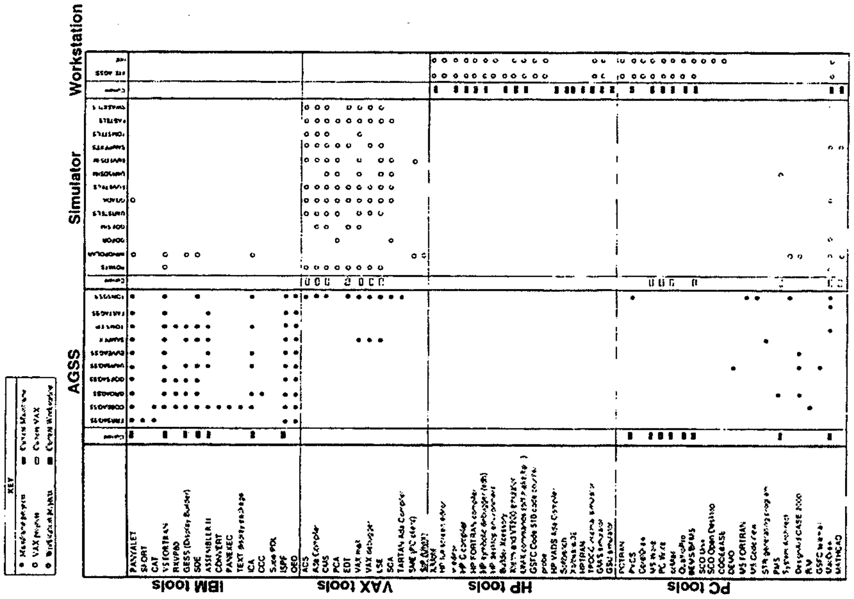
- Based upon a set of 66 services needed for environment frameworks
- Services grouped by convenience into related categories:
  - *Object Management Services.* For creation of data objects and management of the data repository.
  - *Process Management Services.* For definition of computer-assisted software development activities.
  - *Communication Service.* For communication among components of the environment.
  - *User Interface Services.* For communication with users of the framework.
  - *Tool Services.* For installing tools to tailor the framework for specific applications.
  - *Policy Enforcement Services.* For providing security and integrity monitoring services.
  - *Framework Administration and Configuration Services.* For controlling access and resources available in the framework.

PSE Reference Model

- Developed 1991–93 by Navy NGCR program
- It is an end user service model built on top of the NIST–ECMA infrastructure
- Developed for the software engineering software development domain
- Service Categories:
  - Technical Engineering (Requirements, Design, Coding, Traceability, Testing, ...)
  - Technical Management (Configuration management, ...)
  - Project Management (Planning, Estimation, Scheduling, ...)
  - Support service (Editing, Publishing, Figure processing, Email, ...)
- Goal is to map current tool use using this model and then develop a model oriented toward FDD application domain

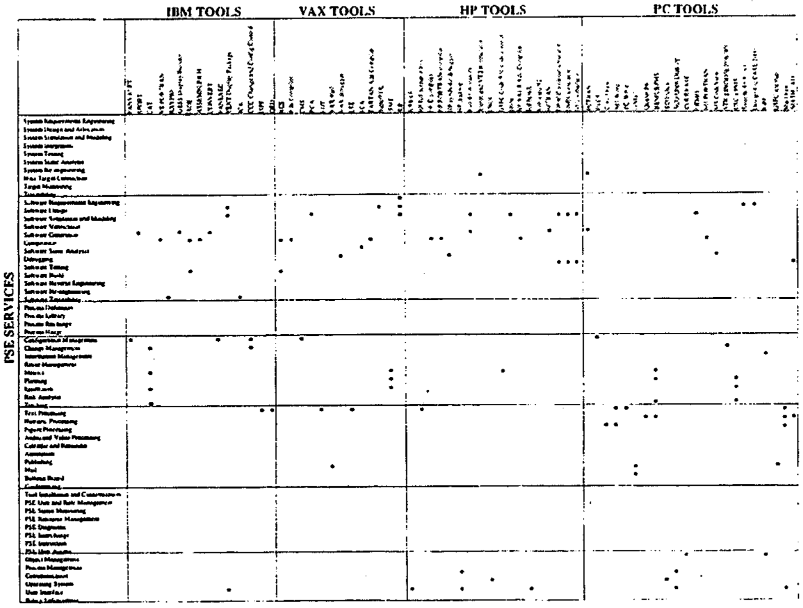
3. What tools are used in the SEL environment?

- Environment evolution over past 20 years:
  - IBM mainframe
  - VAX
  - Workstation-based
- Can we understand what tools have been used in the FDD domain?
- How has tool use changed over the past 20 years?
- Can we map the reference model services to these tools?



Workstation
Simulator
AGSS
KEY
● Used in AGSS
○ Used in Simulator
■ Used in Workstation
IBM tools
VAX tools
HP tools
PC tools
PANVALET
SFORT
CAT
VS FORTRAN
REVPRO
GESS (Display Builder)
SDE
ASSEMBLER II
CONVERT
PANEXEC
ICA
CCC
ISPF
OED
ACS
Ada Compiler
CMS
PCA
EDT
VAX mail
VAX debugger
LSE
SCA
TARTAN Ada Compiler
HP FORTRAN compiler
HP VADS Ada Compiler
GSFC Code 510 case tools
FORTRAN
PVCS
WordPerfect
PC Write
SCO Open Desktop
VS FORTRAN
MATHCAD

Services Used



IBM TOOLS
VAX TOOLS
HP TOOLS
PC TOOLS
PSE SERVICES

Summary of Services Used

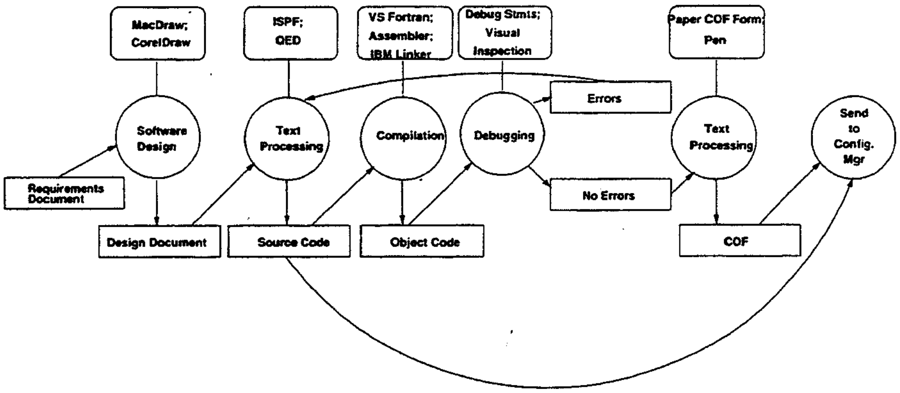
PSE Service	IBM	VAX	HP	PC
System Requirements Engineering				
System Design and Allocation				
System Simulation and Modeling				
System Integration				
System Testing				
System Static Analysis				
System Re-engineering				
Host-Target Connection			•	•
Target Monitoring				
Traceability				
Software Requirements Engineering		•		
Software Design	•	•		•
Software Simulation and Modeling	•	•	•	•
Software Verification				
Software Generation	•		•	•
Compilation	•	•	•	•
Software Static Analysis		•		
Debugging		•	•	•
Software Testing			•	
Software Build	•	•		
Software Reverse Engineering				
Software Re-engineering				
Software Traceability	•			
Process Definition				
Process Library				
Process Exchange				
Process Usage				
Configuration Management	•	•		•
Change Management	•			•
Information Management				•
Reuse Management				
Metrics	•	•	•	•
Planning		•		•
Estimation	•	•		
Risk Analysis				
Tracking	•			•
Text Processing	•	•	•	•
Numeric Processing				•
Figure Processing				•
Audio and Video Processing				
Calendar and Reminder				
Annotation				
Publishing				
Mail		•		•
Bulletin Board				•
Conferencing				
Tool Installation and Customization				
PSE User and Role Management				
PSE Status Monitoring				
PSE Resource Management				
PSE Diagnostic				
PSE Interchange				
PSE Instruction				
PSE User Access				
Object Management				•
Process Management				
Communication			•	•
Operating System			•	•
User Interface	•		•	•
Policy Enforcement				

PSE Service	AGSS	Simulator	Workstation
System Requirements Engineering			
System Design and Allocation			
System Simulation and Modeling			
System Integration			
System Testing			
System Static Analysis			
System Re-engineering			
Host-Target Connection			•
Target Monitoring			
Traceability			
Software Requirements Engineering			
Software Design			
Software Simulation and Modeling			•
Software Verification			
Software Generation	•		•
Compilation	•	•	•
Software Static Analysis		•	
Debugging		•	•
Software Testing			•
Software Build	•	•	
Software Reverse Engineering			
Software Re-engineering			
Software Traceability	•		
Process Definition			
Process Library			
Process Exchange			
Process Usage			
Configuration Management	•	•	•
Change Management			
Information Management			
Reuse Management			
Metrics	•	•	•
Planning	•	•	•
Estimation	•	•	
Risk Analysis			
Tracking	•		•
Text Processing	•	•	•
Numeric Processing	•	•	•
Figure Processing	•	•	•
Audio and Video Processing			
Calendar and Reminder			
Annotation			
Publishing			
Mail	•	•	•
Bulletin Board	•	•	•
Conferencing			
Tool Installation and Customization			
PSE User and Role Management			
PSE Status Monitoring			
PSE Resource Management			
PSE Diagnostic			
PSE Interchange			
PSE Instruction			
PSE User Access			
Object Management			
Process Management			
Communication			•
Operating System			•
User Interface	•		•
Policy Enforcement			

Tool Use in SEL

- Basic functionality of tools has not changed much over last 20 years.
- Not much variability in tool use among different projects
- Underlying technology certainly has changed
- What is relationship of tools to the services they provide?
- Process complexity much more complex than represented by tool use (e.g., New module development on mainframe)

New Module Development Process



Process Complexity?

- Each process can be represented as a graph
- Previous research on graph complexity – *Prime Program Structural Complexity*
  - Services enacted gives a measure of the nodes of the graph, and external data objects gives a measure of the data in the graph.
  - Measure information theoretic content (entropy) of the graph
- Need to develop process models for additional steps in the SEL development process

4. How do we classify FDD applications?

- Current processes are driven by manual intervention
- Has the FDD improved their development practices?
- Can we improve the degree of automation in the FDD environment?
- What are the set of services that define this environment?
  - Three approaches towards applications: Standard development, C++ class library (GSS), FDDS development
  - Study each to look for a common thread of services
  - Develop FDD reference model that more closely defines this application domain

Flight Dynamics "Services" *(Preliminary)*

Database	Telemetry Data	Attitude History
	Measurement Data	Estimation Data
	Uplink Tables	Antenna Contact Parameters
	Instrument View Timeline	
Dynamics	Disturbance	Attitude Dynamics
	3-Axis Stabilized Attitude	Spin-Stabilized Attitude
	Orbit Dynamics	Orbit Model
	Reference Coordinate System	
Hardware	Sensor Model	Actuator Model
Spacecraft Model	Spacecraft Structure	Spacecraft Component
	Surface	
Environment	Stars	Solar System Body
	Earth Atmospheric Density	Geomagnetic Field
	Solar Flux	Gravitational Field
Utilities	Interpolator	Integrator
	Inertial Coordinate Converter	Root-Searcher
	Interval Root Estimator	
Application Domain	Simulation	Estimation
Telemetry		
Measurement	Sensor Measurements	Actuator Measurements
	Star Identification	Validation
Estimation	Estimator	Dynamic Model
	Measurement Model	
Simulation	Schedule	Event
	Condition	Command
	Simulated Sensor	Simulated Actuator

Model Measurements

- This represents the first steps toward quantifying this model. Parameters:
  - Percent reuse is a first approximation of degree of automation
  - Services implemented provides a rough approximation of level of abstraction
  - Entropy (prime program structural complexity) provides a first approximation of process complexity
- Current model is static. Need to factor in annual increase in technological changes. (e.g., If average increase is 4%, what does this even mean?)
- Reliability of development process (i.e., faults found via SEL database) and productivity data provide some validation of process complexity
- Goal: To develop a measurement process using these attributes with validation of this model through SEL database

Conclusions

- This talk represents a first step at addressing the important issue of measuring process complexity
- The goal is compounded by the knowledge that "average" complexity increases over time
- Within SEL, tool use has changed over time, but tool functionality has not changed significantly
- Do not have a good "figure of merit" for effective tool use
- We are currently working on developing such a model
- An application-specific model of environmental services should help

omit
this
page

Session 4: Models

Reliability and Risk Analysis of the NASA Space Shuttle Flight Software
Norman Schneidewind, Naval Postgraduate School

Modeling and Simulation of Software Projects
Anke Drappa, University of Stuttgart

Evaluating Empirical Models for the Detection of High-Risk Components: Some Lessons Learned
Filippo Lanubile, University of Bari

Reliability and Risk Analysis of the NASA Space Shuttle Flight Software

Norman F. Schneidewind

Code SM/Ss
Naval Postgraduate School
Monterey, CA 93943

Voice: (408) 656-2719
Fax : (408) 656-3407
Internet: schneidewind@nps.navy.mil

51-61
415607
360701
22P.

Introduction

We have used two categories of software reliability measurements and predictions in combination to *assist* in assuring the safety of the software of the *NASA Space Shuttle Primary Avionics Software System*. The two categories are: 1) measurements and predictions that are associated with residual software faults and failures, and 2) measurements and predictions that are associated with the ability of the software to survive a mission without experiencing a serious failure. In the first category are: *remaining failures, total failures, fraction of remaining failures,* and *test time required to attain a given number or fraction of remaining failures*. In the second category are: *time to next failure* and *test time required to attain a given time to next failure*. In addition, we define the risk associated with not attaining the required *remaining failures and time to next failure*. Lastly, we have derived a quantity from the *fraction of remaining failures* that we call *operational quality*. The benefits of predicting these quantities are: 1) they provide confidence that the software has achieved safety goals, and 2) they provide a means of rationalizing how long to test a piece of software. Having predictions of the extent that the software is not fault free (*remaining failures*) and its ability to survive a mission (*time to next failure*) are meaningful for assessing the risk of deploying safety critical software. In addition, with this type of information a program manager can determine whether more testing is warranted, or whether the software is sufficiently tested to allow its release or unrestricted use. These predictions, in combination with other methods of assurance, such as inspections, defect prevention, project control boards, process assessment, and fault tracking, provide a quantitative basis for achieving safety and reliability objectives [BIL94].

Loral Space Information Systems, the primary contractor on the *Shuttle Flight Software* project is experimenting with a promising algorithm which involves the use of the *Schneidewind Software Reliability Model* to compute a parameter: *fraction of remaining failures=remaining failures/maximum failures* as a function of the archived failure history during testing and operation. [KEL95]. Our prediction methodology provides bounds on *test time*, *remaining failures*, *operational quality*, and *time to next failure* that are necessary to meet *Shuttle* software safety requirements. We also show that there is a pronounced asymptotic characteristic to the *test time* and *operational quality* curves that indicate the possibility of big gains in reliability as testing continues; eventually the gains become marginal as testing continues. We conclude that the prediction methodology is feasible for the *Shuttle* and other safety critical applications.

Although *remaining failures* has been discussed in general as a type of software reliability prediction [MUS87], and various stopping rules for testing have been proposed, based on the economics of testing [DAL94] and a testability criterion [VOA95], our approach is novel because we integrate safety criteria, risk analysis, and a stopping rule for testing. Furthermore, we use reliability measurements and predictions to assess whether safety goals are likely to be achieved. Thus we advocate using safety analysis and reliability analysis synergistically in a mutually supportive way rather than treat these fields as disjoint and unrelated.

Criteria for Safety

If we define our safety goal as the reduction of failures that would cause loss of life, loss of mission, or abort of mission to an acceptable level of risk [LEV86], then for software to be ready to deploy, after having been tested for time t_2, we must satisfy the following criteria:

1) predicted remaining failures $R(t_2)<R_c$, (1)
where R_c is a specified critical value , and

2) predicted time to next failure $T_F(t_2)>t_m$, (2)
where t_m is mission duration.

For systems that are tested and operated continuously like the *Shuttle*, t_2, $T_F(t_2)$, and t_m are measured in execution time. Note that, as with any methodology for assuring software safety, we can't guarantee safety. Rather, with these criteria, we seek to reduce the risk of deploying the software to an acceptable level.

Remaining Failures Criterion

On the assumption that the faults associated with failures are removed (this is the case for the *Shuttle*), *criterion 1* specifies that the residual failures and faults must be reduced to a level where the risk of operating the software is acceptable. As a practical matter, we suggest $R_c=1$. That is, the goal would be to reduce the expected remaining failures to less than one before deploying the software. If we predict $R(t_2)\geq R_c$, we would continue to test for a total time $t_2'>t_2$ that is predicted to achieve $R(t_2')<R_c$, on the assumption that we will experience more failures and correct more faults so that the remaining failures will be reduced by the quantity $R(t_2)-R(t_2')$. If the developer does not have the resources to satisfy the criterion or is unable to satisfy the criterion through additional testing, the risk of deploying the software prematurely should be assessed (see the next section). We know from Dijkstra's dictum that we can't demonstrate the absence of faults; however we can reduce the risk of failures occurring to an acceptable level, as represented by R_c. This scenario is shown in Figure 1. In *case A* we predict $R(t_2)<R_c$ and the mission begins at t_2. In *case B* we predict $R(t_2)\geq R_c$ and postpone the mission until we test for time t_2' and predict $R(t_2')<R$. In both cases *criterion 2)* must also be satisfied for the mission to begin.

One way to specify R_c is by failure severity level (e.g., *severity level 1* for life threatening failures). Another way, which imposes a more demanding safety requirement, is to specify that R_c represents *all* severity levels. For example, $R(t_2)<1$ would mean that $R(t_2)$ must be less than one failure, *independent* of severity level.

Time to Next Failure Criterion

Criterion 2 specifies that the software must survive for a time greater than the duration of the mission. If we predict $T_F(t_2) \le t_m$, we would continue to test for a total time $t_2">t_2$ that is predicted to achieve $T_F(t_2")>t_m$, on the assumption that we will experience more failures and correct more faults so that the time to next failure will be increased by the quantity $T_F(t_2")-T_F(t_2)$. Again, if it is infeasible for the developer to satisfy the criterion for lack of resources or failure to achieve test objectives, the risk of deploying the software prematurely should be assessed (see the next section). This scenario is shown in Figure 2. In *case A* we predict $T_F(t_2)>t_m$ and the mission begins at t_2. In *case B* we predict $T_F(t_2) \le t_m$ and postpone the mission until we test for time $t_2"$ and predict $T_F(t_2")>t_m$. In both cases *criterion 1)* must also be satisfied for the mission to begin. If neither criterion is satisfied, we test for a time which is the greater of t_2' or $t_2"$.

Risk Assessment

The amount of test execution time t_2 can be considered a measure of the maturity of the software. This is particularly the case for systems like the *Shuttle* where the software is subjected to continuous and rigorous testing for several years. If we view t_2 as an input to a risk reduction process, and $R(t_2)$ and $T_F(t_2)$ as the outputs, we can portray the process as shown in Figure 3, where R_c and t_m are shown as "levels" of safety that control the process.

Remaining Failures

We can formulate the risk of *criterion 1* as follows:

$$\text{Risk } R(t_2)= (R(t_2)-R_c)/R_c=(R(t_2)/R_c)-1 \qquad (3)$$

We plot equation (3) in Figure 4 as a function of t_2 for $R_c=1$, where *positive*, *zero*, and *negative* risk correspond to $R(t_2)>R_c$, $R(t_2)=R_c$, and $R(t_2)<R_c$, respectively, and the *UNSAFE* and *SAFE* regions are above and below the X-axis, respectively. This graph is for the *Shuttle operational increment OID;* an operational increment (OI) is comprised of modules and configured from a series of builds to meet mission functional requirements. In this example we see that at approximately $t_2=57$ the risk transitions from the *UNSAFE* region to the *SAFE* region.

Time to Next Failure

Similarly, we can formulate the risk of *criterion 2* as follows:

$$\text{Risk } T_F(t_2)=(t_m-T_F(t_2))/t_m=1-(T_F(t_2))/t_m \qquad (4)$$

We plot equation (4) in Figure 5 as a function of t_2 for $t_m=8$ days (a typical mission duration time for this OI), where *positive*, *zero*, and *negative* risk corresponds to $T_F(t_2)<t_m$, $T_F(t_2)=t_m$, and $T_F(t_2)>t_m$, respectively, and the *UNSAFE* and *SAFE* regions are above and below the X-axis, respectively. This graph is for the *Shuttle operational increment OIC*. In this example we see that at all values of t_2 the risk is in the *SAFE* region.

Approach to Prediction

In order to support our safety goal and to assess the risk of deploying the software, we make various reliability and quality predictions. In addition, we use these predictions to make tradeoff analysis between reliability and test time (cost). Thus our approach to reliability prediction is the following: 1) Use a software reliability model to predict *total failures*, *remaining failures*, and *operational quality*; 2) Predict the *time to next failure* (beyond the last observed failure); 3) Predict the *test time* necessary to achieve required levels of *remaining failures* (fault) level, *operational quality*, and *time to next failure;* and 4) Examine the tradeoff between increases in levels of reliability and quality with increases in testing.

The predictions are based on the *Schneidewind Software Reliability Model*, one of the four models recommended in the *AIAA Recommended Practice for Software Reliability* [AIA93]. It is not our purpose to derive the model equations because they have been derived in other publications [AIA93, SCH93, SCH92, SCH75]. Rather we apply the model to analyze the reliability of the *Space Shuttle Primary Avionics Software*.

Because the flight software is run continuously, around the clock, in simulation, test, or flight, "time" refers to continuous execution time and test time refers to execution time that is used for testing.

Making Safety Decisions

In making the decision about how long to test t_2, we apply our safety criteria and risk assessment approach. We use Table 1 and Figure 6 to illustrate the process. For *test time* t_2=18 (when the last failure occurred on *OIA*), R_c=1, and t_m=8 days (.267 intervals), we show *remaining failures* $R(t_2)$, *risk of remaining failures, time to next failure* $T_F(t_2)$, *risk of time to next failure*, and *operational quality* Q, where Q=1-*fraction remaining failures*, in Table 1. These results indicate that safety *criterion 2* is satisfied but not *criterion 1* (i.e., *UNSAFE* with respect to remaining failures); also *operational quality* is low. With these results in hand, the software manager could choose to continue to test. If testing were to continue until t_2=52, the predictions in Table 1 and annotated on Figure 6 would be obtained. These results show that *criterion 1* is now satisfied *(i.e., SAFE)* and *operational quality* is high. We also see that at this value of t_2, further increases in t_2 would not result in a significant increase in reliability and safety.

Table 1
Safety Criteria Assessment

		R_c=1	t_m=8 days		
t_2	$R(t_2)$	RiskR(t2)	$T_F(t_2)$	$RiskT_F(t_2)$	Q
18	4.76	3.76	3.87	-13.49	.60
52	0.60	-.40	*	*	.95

t_2: 30 day intervals
* Can't predict because predicted Remaining Failures is less than one.

Conclusions

Software reliability models provide one of several tools that software reliability managers of the *Space Shuttle Primary Avionics Software* are using to provide confidence that the software meets required safety goals. Other tools are inspections, software reviews, testing, change control boards, and perhaps most important -- experience and judgement. We have shown how to apply these models; the approach would seem to be applicable to other safety critical systems. We encourage practitioners to apply these methods.

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.

[BIL94] C. Billings, et al, "Journey to a Mature Software Process", IBM Systems Journal Vol. 33, No. 1, 1994, pp. 46-61.

[DAL94] S. R. Dalal and A. A. McIntosh, "When to Stop Testing for Large Software Systems with Changing Code", IEEE Transactions on Software Engineering, Vol. 20, No. 4, April 1994, pp. 318-323.

[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.

[LEV86] Nancy G. Leveson, "Software Safety: What, Why, and How", ACM Computing Surveys, Vol. 18, No. 2, June 1986, pp. 125-163.

[MUS87] John Musa, et al, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, New York, 1987.

[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.

[SCH75] Norman F. Schneidewind, "Analysis of Error Processes in Computer Software", Proceedings of the International Conference on Reliable Software, IEEE Computer Society, 21-23 April 1975, pp. 337-346.

[VOA95] Jeffrey M. Voas and Keith W. Miller, "Software Testability: The New Verification", IEEE Software, Vol. 12, No. 3, May 1995, pp. 17-28.

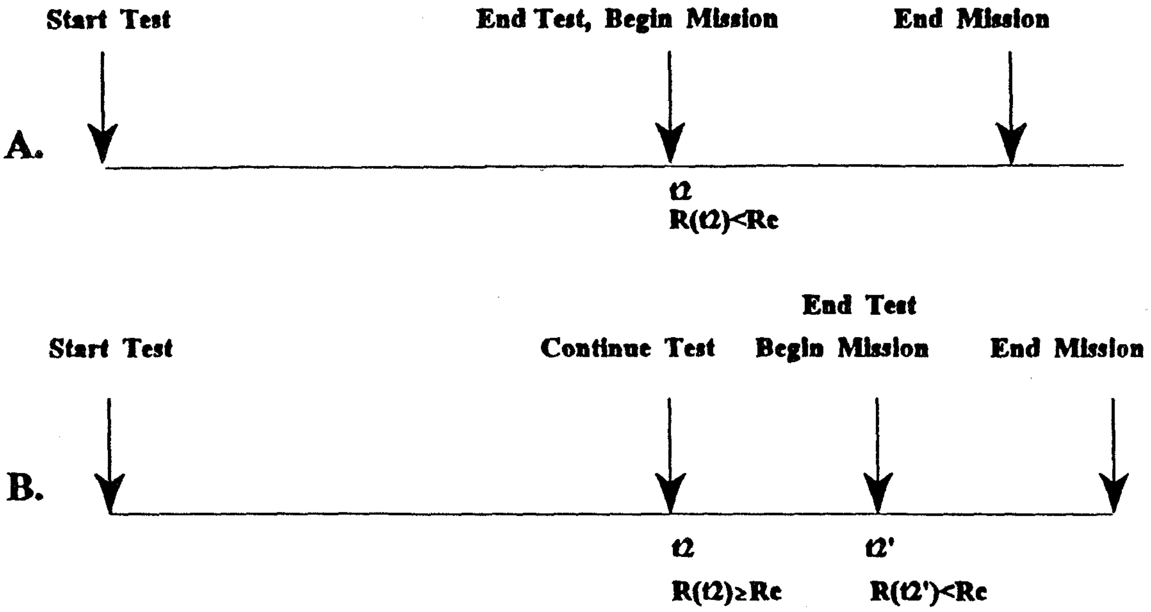


Figure 1. Remaining Failures Criterion Scenario

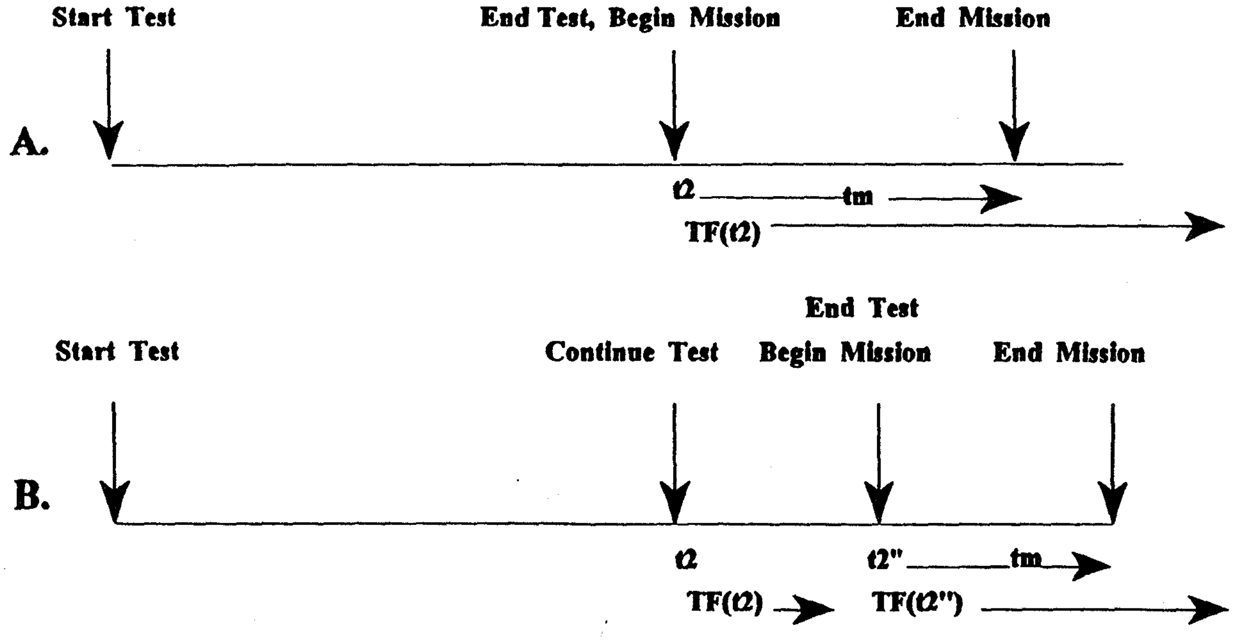


Figure 2. Time to Next Failure Criterion Scenario

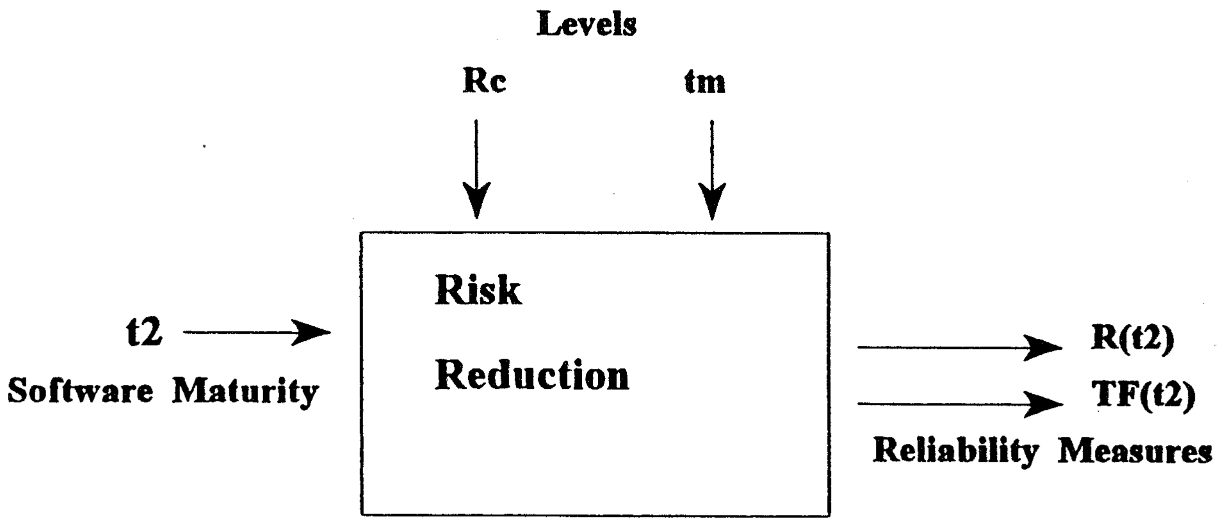


Figure 3. Risk Reduction

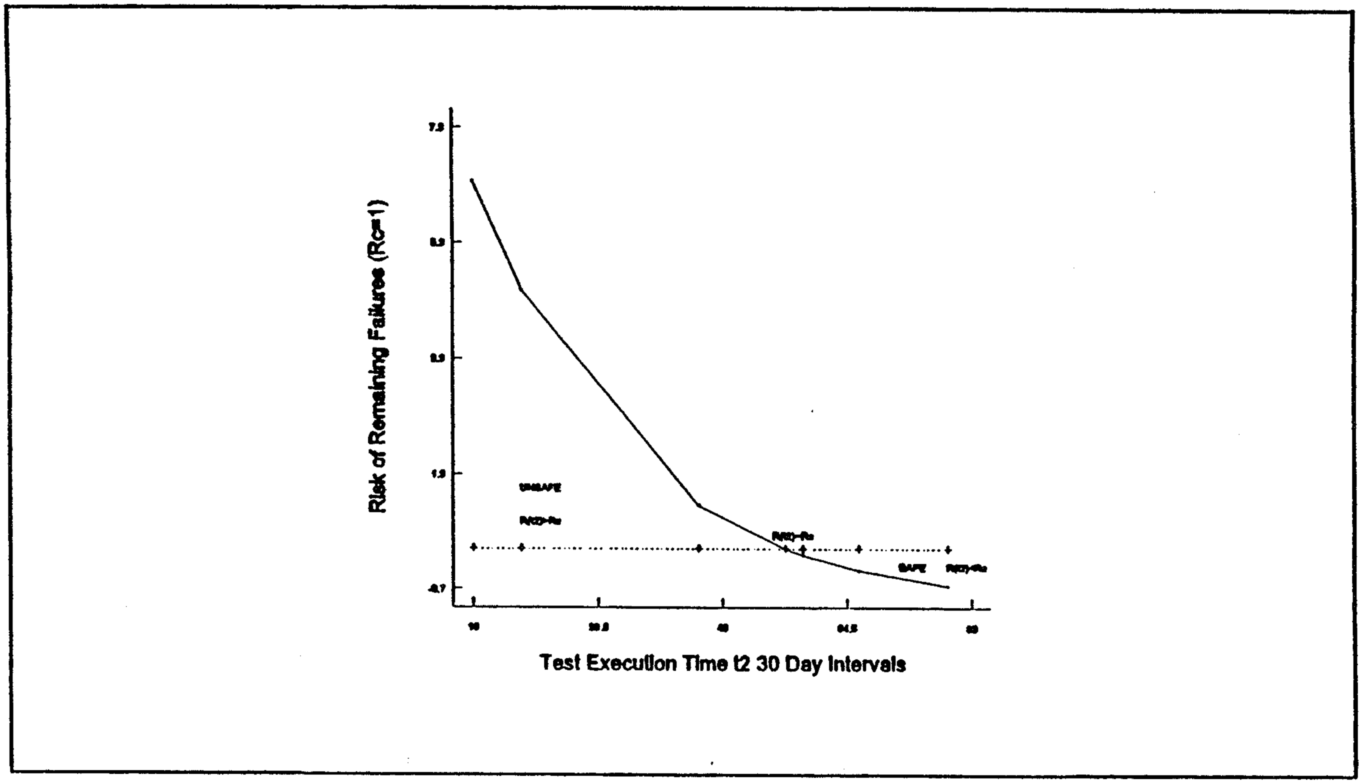


Figure 4. Risk of Remaining Failures

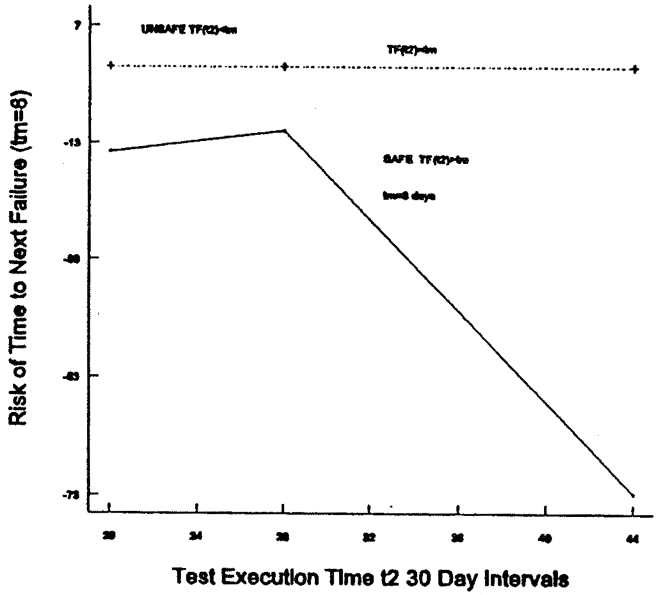


Figure 5. Risk of Time to Next Failure

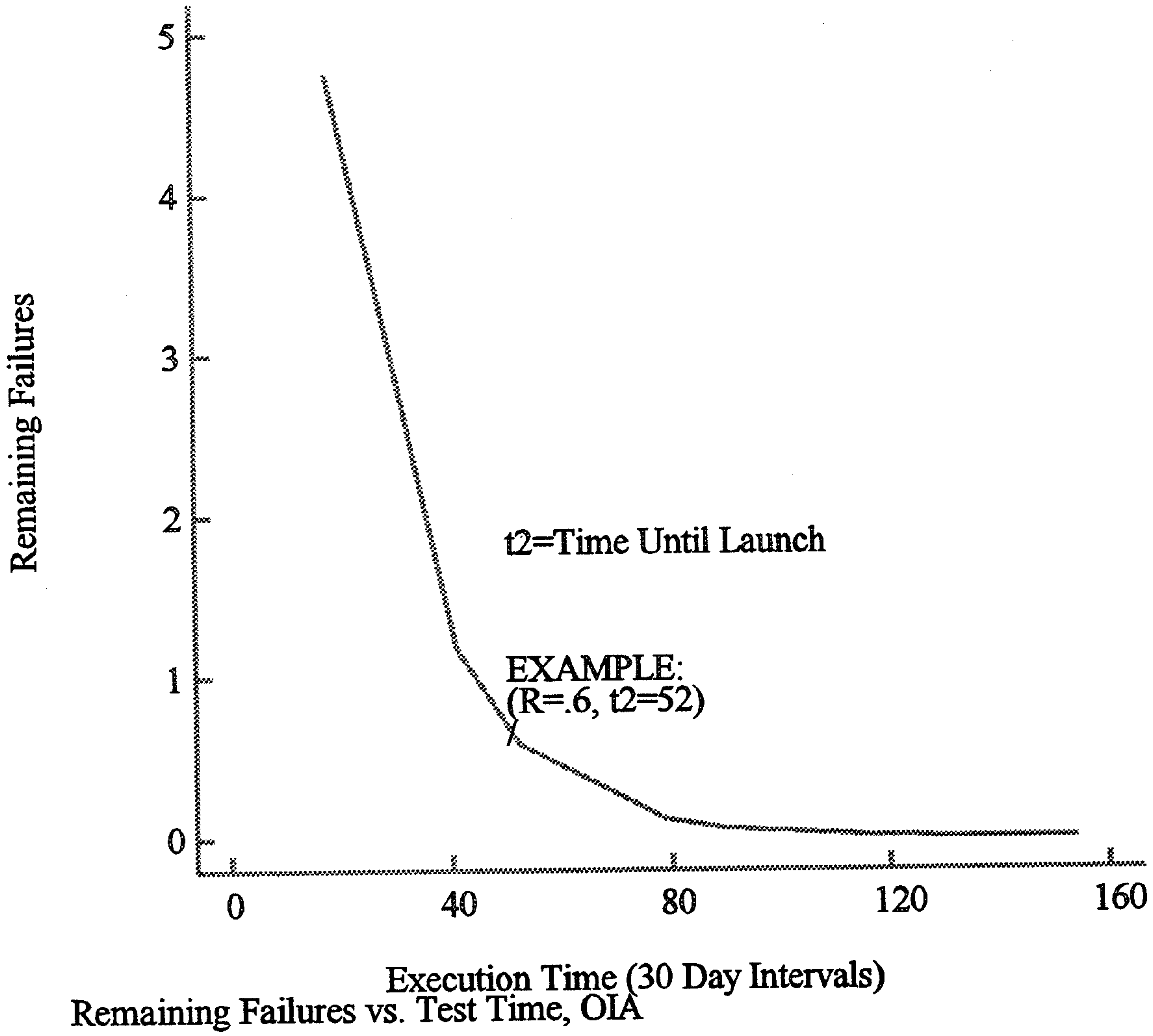


Figure 6. Launch Decision: Remaining Failures versus Test time

Reliability and Risk Analysis of the NASA Space Shuttle Flight Software

Norman F. Schneidewind

Code SM/Ss
Naval Postgraduate School
Monterey, CA 93943

Voice: (408) 656-2719
Fax : (408) 656-3407

Internet:
schneidewind@nps.navy.mil

OUTLINE

o *Shuttle* Critical Software

o *Shuttle* Software Configuration

o Safety Criteria

- Remaining Failures

- Time to Next Failure

o Risk Reduction

- Risk of Remaining Failures

- Risk of Time to Next Failure

o Software Deployment Decision

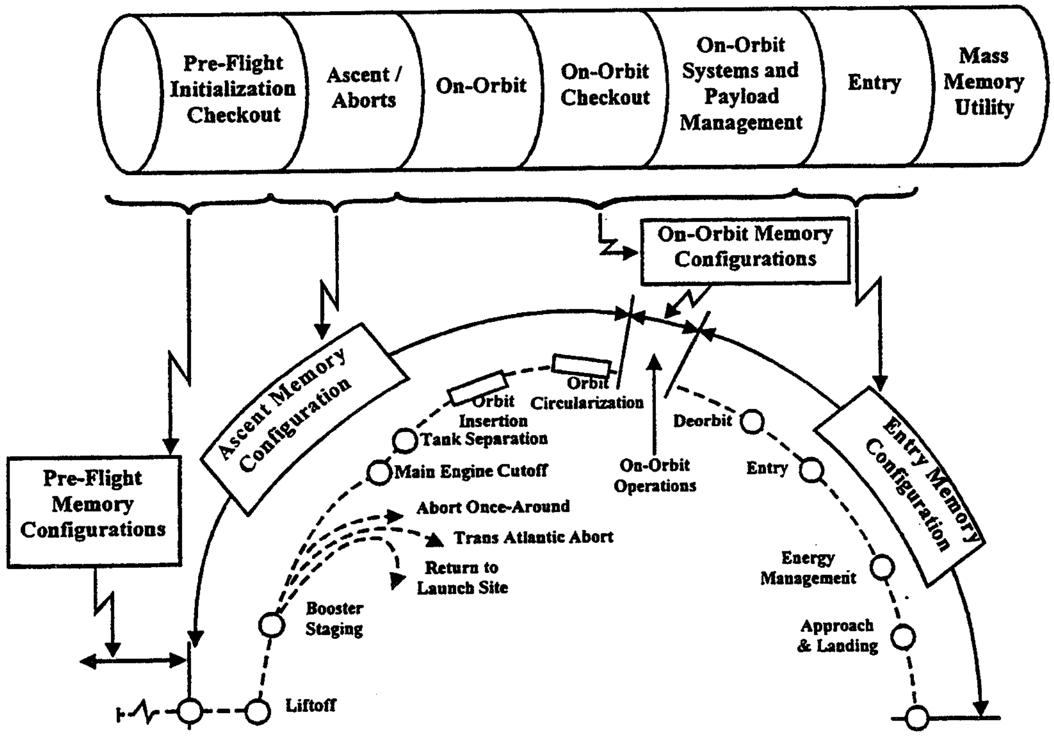
o Conclusions

SOFTWARE CRITICAL TO NASA SHUTTLE

ONBOARD DATA PROCESSING SYSTEM AND SOFTWARE

- Mission - Critical Software System - "Fly By Wire"
- Supports All Flight Phases From Pre - Launch to Rollout
- Hardware: 5 General Purpose Computers, Associated Displays, Keyboards, Mass Storage Devices
- Software:
  - Primary Avionics Software System (PASS) Keyed to Mission Phases
  - Backup Flight System Provides Redundancy During Critical Ascent and Entry Phases
- During Critical Phases, 4 of the 5 Computers Execute the PASS Redundantly

SOFTWARE CONFIGURATION FOR A TYPICAL SHUTTLE FLIGHT



Pre-Flight Initialization Checkout

Ascent / Aborts

On-Orbit

On-Orbit Checkout

On-Orbit Systems and Payload Management

Entry

Mass Memory Utility

On-Orbit Memory Configurations

Pre-Flight Memory Configurations

Ascent Memory Configuration

Entry Memory Configuration

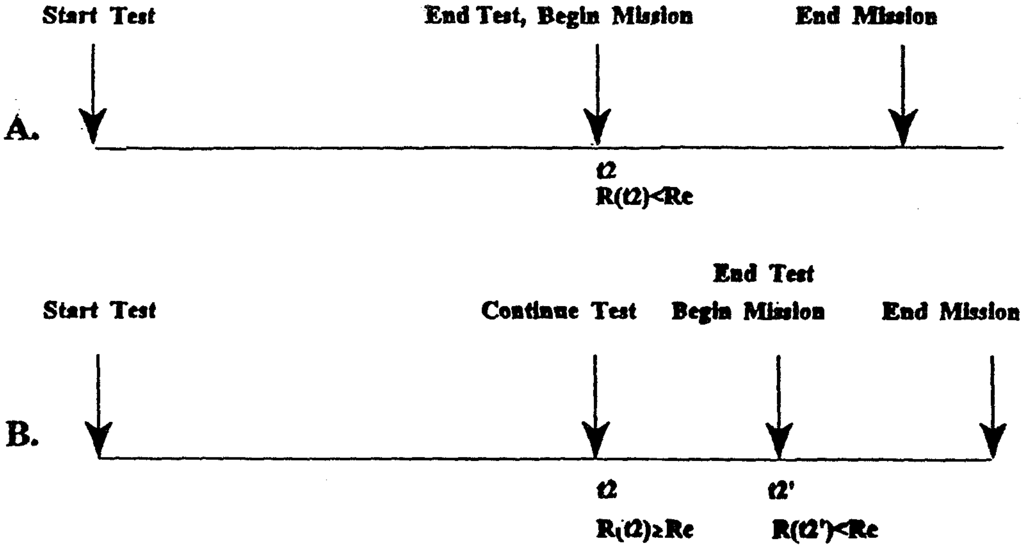


Figure 1. Remaining Failures Criterion Scenario

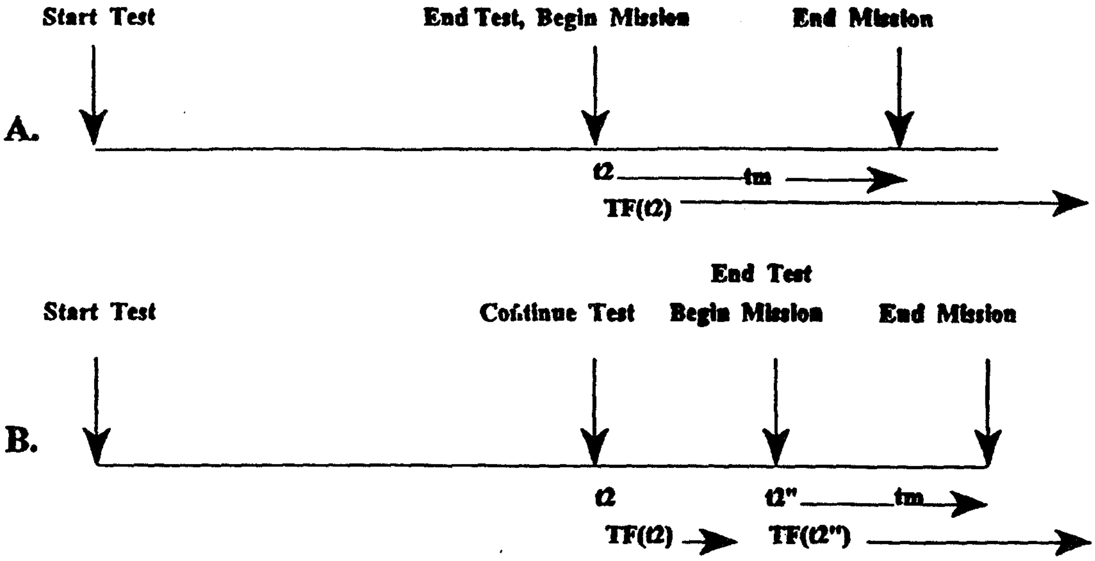


Figure 2. Time to Next Failure Criterion Scenario

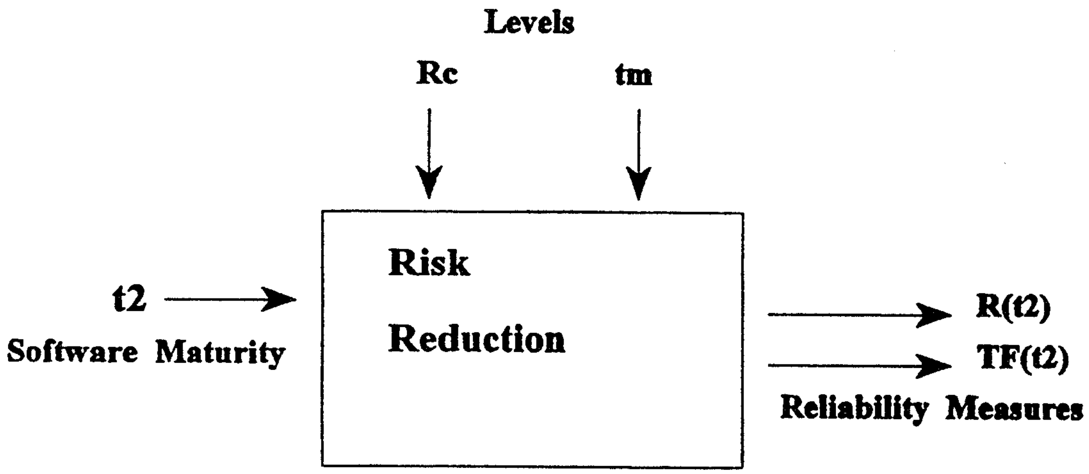
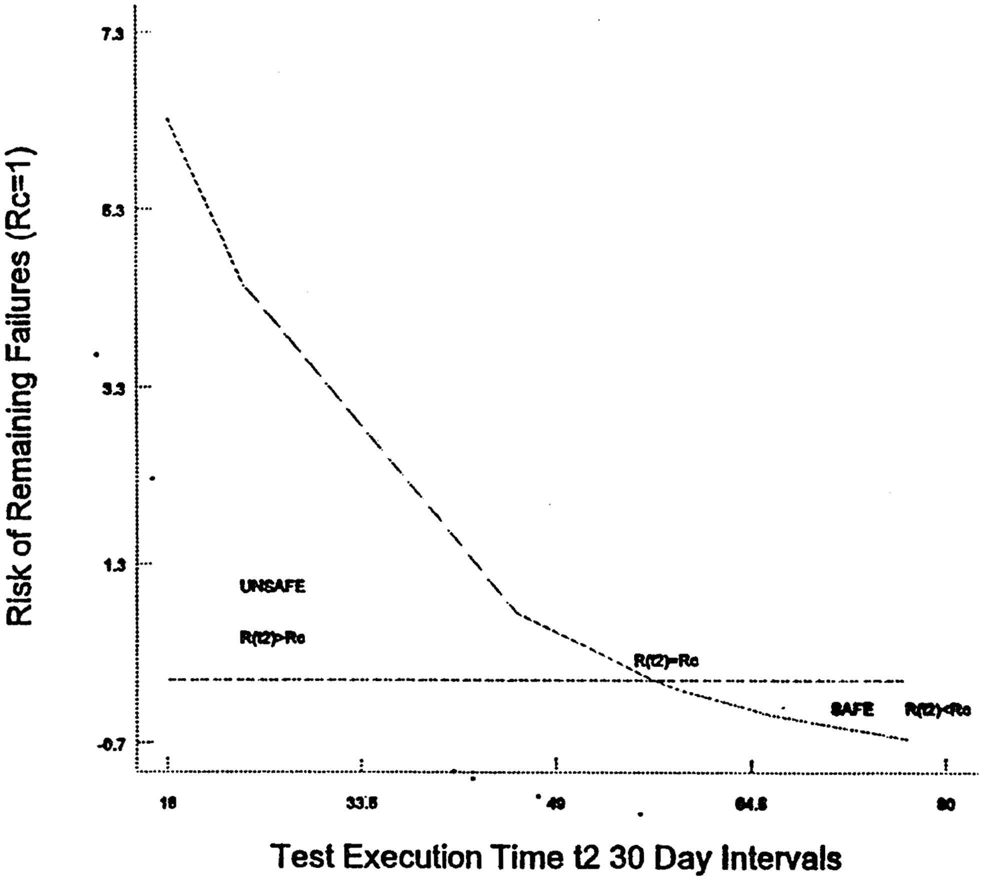
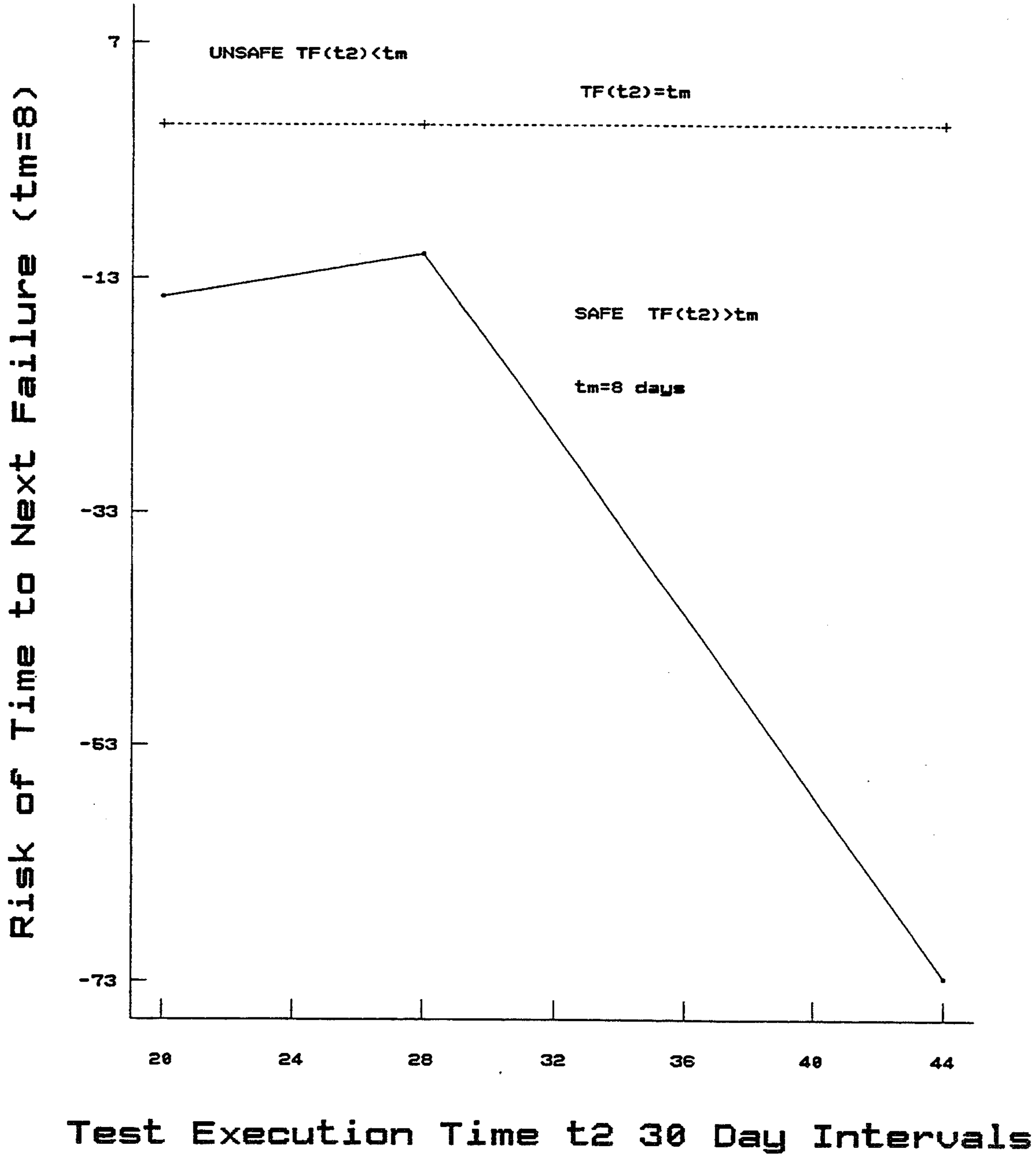


Figure 3. Risk Reduction


Risk of Remaining Failures (Rc=1)
7.3
5.3
3.3
1.3
-0.7
UNSAFE
R(t2)>Rc
R(t2)=Rc
SAFE
R(t2)<Rc
18
33.5
49
64.5
80
Test Execution Time t2 30 Day Intervals


UNSAFE TF(t2)<tm
TF(t2)=tm
SAFE TF(t2)>tm
tm=8 days
7
-13
-33
-53
-73
Risk of Time to Next Failure (tm=8)
20
24
28
32
36
40
44
Test Execution Time t2 30 Day Intervals

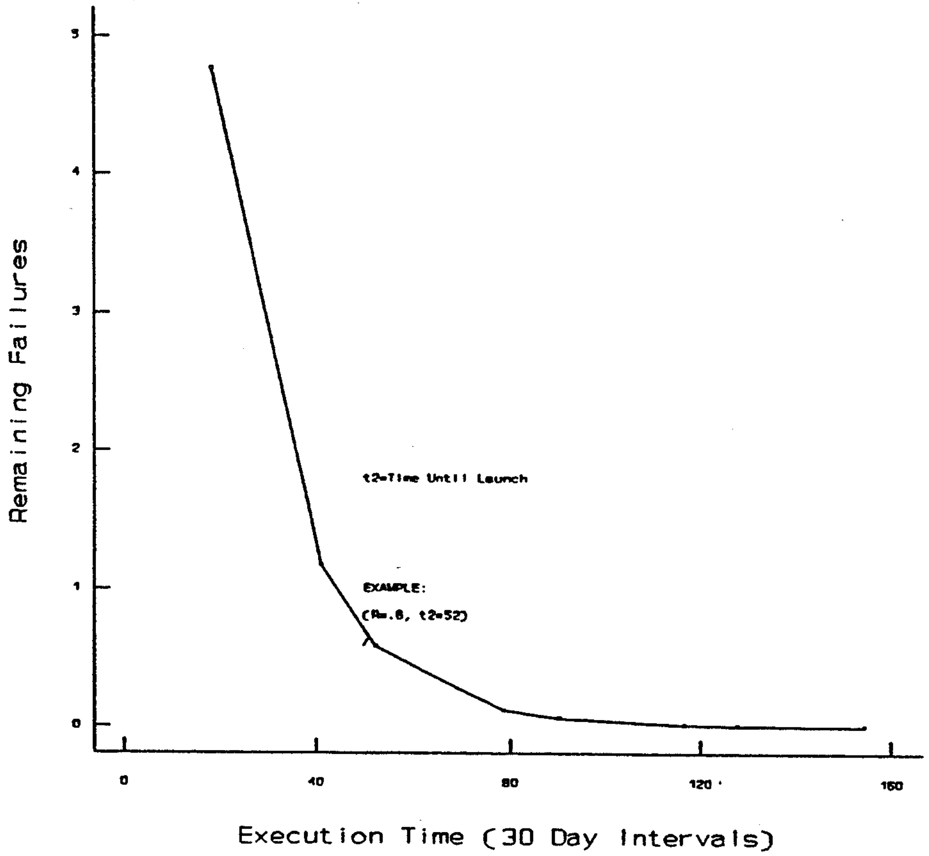
SAFETY CRITERION 1:

The first criterion required for flight safety is that the predicted ***remaining failures*** (i.e., residual faults) be at an acceptable level (e.g., $R(t_2)<1$) at ***test time*** t_2.

This is shown in the figure where we test for a time t_2=***time until launch***=52 intervals, and at this time R(52)=.6.

SAFETY CRITERION 2:

Not shown is the second criterion: $T_F(52)>t_m$(***time to next failure>mission duration***).



Remaining Failures vs. Test Time, OIA

Conclusions

Software reliability models provide one of several tools that software reliability managers of the *Space Shuttle Primary Avionics Software* are using to provide confidence that the software meets required safety goals. Other tools are inspections, software reviews, testing, change control boards, and perhaps most important -- experience and judgement. We have shown how to apply these models; the approach would seem to be applicable to other safety critical systems. We encourage practitioners to apply these methods.

WHERE TO OBTAIN METRICS AND RELIABILITY DOCUMENTS:

STANDARD FOR A SOFTWARE QUALITY METRICS METHODOLOGY, IEEE STD 1061-1992, MARCH 12, 1993:

IEEE Standards Office
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331, U.S.A.
Tel: 1-800-678-IEEE

STATISTICAL MODELING AND ESTIMATION OF RELIABILITY FUNCTIONS FOR SOFTWARE (SMERFS):

Version 5 of this program is available from Dr. William H. Farr, Code B10, Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA 22448, U.S.A.
Tel: 1-540-653-8388.

ANSI/AIAA RECOMMENDED PRACTICE FOR SOFTWARE RELIABILITY:

James French
Director of Standards
AIAA Headquarters
370 L'Enfant Promenade, SW
Washington, DC 20024-2518, U.S.A.
Tel: 1-202-646-7400

Modeling and Simulation of Software Projects

A. Drappa, M. Deininger, J. Ludewig, R. Melchisedech

Software Engineering Group, University of Stuttgart
Breitwiesenstr. 20 - 22, D-70565 Stuttgart, Germany
drappa@informatik.uni-stuttgart.de

1. Introduction to SESAM

Over the last twenty five years, software development projects have increasingly been plagued by schedule and cost overruns, while product quality is often poor. The term "software crisis" has been coined to highlight this situation. The lack of adequate methods and tools to support software development, and of techniques to simplify project management tasks have been identified as major factors contributing to the "software crisis".
In reaction to this, the software engineering community has attempted to devise a number of analytical and constructive operations to both handle the complexity of software development and assure quality of the resulting products. However, despite an impressive evolution of methods and technology in the past years, neither the software development process nor overall product quality have improved significantly. Part of the reason for this appears to be a serious lack of understanding of the software development process. It is often unclear how to integrate methods and tools in the development process to achieve best results because little is known about the most important influencing factors that determine process and product quality.

Our research project SESAM (Software Engineering Simulation by Animated Models) attempts to address the problems outlined above. With the SESAM project, we pursue the following two main goals.

1) *Building quantitative models of software development projects to gain a better understanding of the underlying processes.*

   Detailed description of software processes is an absolute requirement for informed discussion about effects observable in software development projects. Software projects can be described by a set of simple effects, where each effect influences the course of the process and the quality of the resulting products. The software project is thus determined by simple rules and their effect on the whole process. Our goal is to identify these effects, and to describe them in a quantitative way wherever possible. Even though some of the quantitative effects are currently hypothetical in nature, the formal description provides a sound basis for their validation with empirical data.

2) *Simulation of software development projects, taking the models as a basis to educate future project managers.*

   Teaching project management from text books has been proved to be insufficient, while training on the job is difficult due to the length and costs of software projects. Taking the knowledge captured in SESAM models as a basis, the simulation of software projects allows students to gain reality-like experience without jeopardizing the progress of real projects. Thus, the simulation offers the opportunity to transfer the available knowledge of software processes to project managers who in turn can apply this knowledge in real industrial projects.

The idea of SESAM was first described in 1989 [Ludewig89]. Since then we have built three consecutive prototypes, and have finished the system SESAM-1 in 1994. This work has produced new results in three important categories:

- more elaborate concepts of SESAM models,
- further development of a language to adequately describe those models, and
- new features needed for the simulation system.

A selection of these results will be presented in the following three sections. Finally, we will summarize our experience with a SESAM model in a project management course held at our department and briefly outline some of our current research activities.

2. Basic Principles and Related Work

The approach we took to develop SESAM is influenced by a number of underlying basic concepts. In the following section, we will outline our general approach, refer to some of the concepts developed by other groups and illustrate how our approach differs from previous attempts to model software development processes.

Building models that can be validated.

Effective teaching of software engineering and project management knowledge by simulation requires that the models used be realistic, provide quantitative information, and be capable of reflecting process and product quality. Thus in SESAM, we have to identify cause and effect relationships between objects involved in the software process and to describe them quantitatively as far as possible. Consequently, the SESAM models were designed to be able to reflect and process quantitative data. Since the software engineering knowledge currently available largely is a collection of "rules of thumb", our models have to be bolstered by empirical data from real projects. Therefore, SESAM models are intended to be detailed (fine-grained) so that results of the simulation can be directly related to real project data.

Learning by trial and error.

The traditional way of teaching project management is to convey one (the best) predefined solution of a given problem to a student. In our experience, teaching is more effective, and learning is much easier, if the student has a chance to try different solutions. Subsequently, he or she can analyze which solution is the best, and why. Therefore, software projects should be simulated by allowing the student to trigger any sequence of actions, driving the project in what he or she thinks is the right direction. Conversely, the simulation system reacts by presenting informal messages reflecting the current state of the project. As a consequence, SESAM models are built to be interactive, i.e. models must accept input from the student, and react appropriately to the actions taken.

Related Work

Quantitative modeling is not a new idea. The pioneering work of Tarek Abdel-Hamid aimed at gaining a fundamental understanding of software project management processes [Abdel-Hamid91]. His results have influenced a number of similar approaches [Levary91], [Carr90], the basic idea of which is to describe and simulate software processes using system dynamics. The drawback of system dynamics models is that they are neither interactive, nor fine-grained. While well suited to describe quantitative aspects, they do not provide means of interaction between model and student for training purposes.

At the present time, software process modeling is an active and growing area of research. An overview is presented in [Curtis92]. Among the principal goals of this research is the construction of Process Centered Software Engineering Environments (PCSEE) offering tools to facilitate product development and integrate information about the whole process. By strongly guiding the process, management tasks can be simplified significantly [Snowdon94]. The construction of PCSEEs also requires detailed description of software processes, and much effort has been devoted to the definition of new modeling approaches and languages [Finkelstein94]. These approaches, however, tend to emphasize a predefined course of the modeled project, and do not provide the flexibility required for our purposes.

3. SESAM – Modeling Approach and Language

We concluded that for our purposes, a new modeling approach was desirable. We set out to define a multi-paradigm language integrating a number of well established concepts.

The following description of the SESAM models is structured into two parts. The static perspective mainly focuses on the description and type definition of objects involved in the software process and possible relationships between these objects. The dynamic behavior is described by a set of generic rules specifying actions and effects which change the state of the simulated software project [Schneider94].

Static perspective – the scheme

To describe the static perspective called the SESAM scheme, the extended entity-relationship notation is used. Types of real world objects, like Developer or Document, and of possible relationships between these objects, like reads or writes, are identified and further described by attributes. Possible attributes of the entity type Developer include for example name, age, or experience. The type Document could be characterized by the attributes name, size, and #_of_faults. The scheme is described graphically. A very simple part of the scheme is shown in figure 3.1.

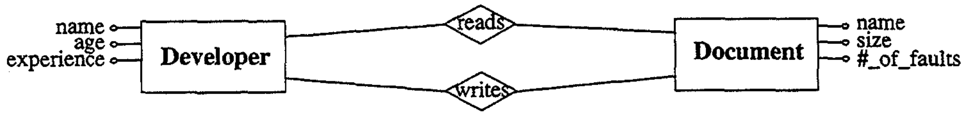


Figure 3.1: Part of a SESAM scheme

Dynamic perspective – the rules

The dynamic behavior is represented by a set of rules. Each rule essentially describes how the state of the simulated project changes with time, or as the result of an action triggered by the project manager. Consider a simple example: The project leader asks a developer to write a document, perhaps the specification. One effect of this action is that as long as the developer is writing the document, the document grows at a speed depending on the developer's experience. Assume further that the average writing productivity is 3 pages per day, and that experience is a multiplier with a value of less than one for a rookie, and greater than one for a very experienced team member. In SESAM this rule is described as shown in figure 3.2.

The graphical notation for SESAM rules is based on a combination of graph grammars and system dynamics. The basic idea is that the current state of the simulated project is represented by a graph to which graph grammar productions can be applied. Each SESAM rule performs

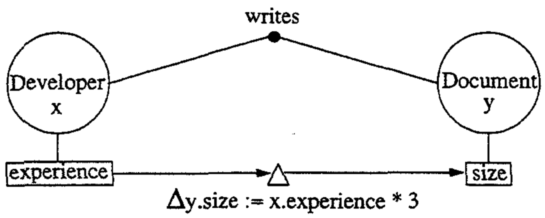


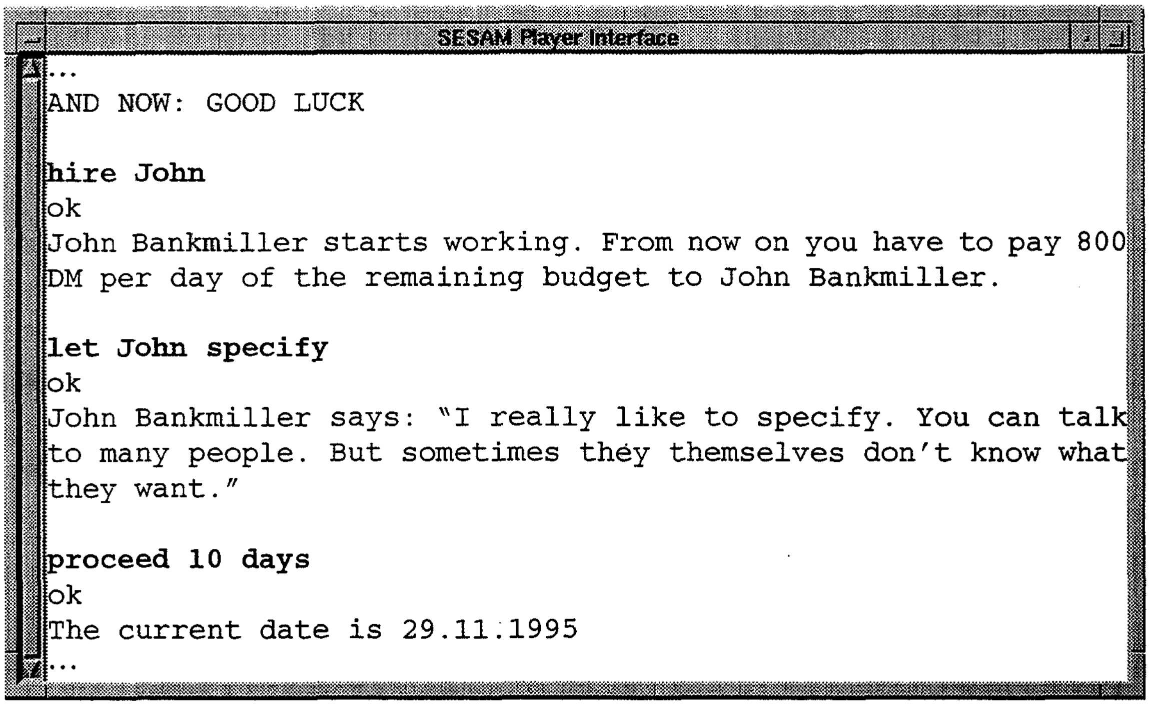
Figure 3.2: A simple SESAM rule

such a graph production, i.e. it specifies which subgraph must be matched so that the rule can be activated. The notation has been extended by system dynamics elements to reflect continuous changes of attribute values with simulation time (see attribute size in figure 3.2). The interaction between model and player is described by events. Rule notation and the formal foundations are discussed in more detail in [Drappa95].

Before simulation can begin, another model component has to be provided: the initial situation. This part essentially defines the initial graph to which the rules can be applied. The initial situation comprises the problem description of a specific project, and instances from the entity or relationship types defined in the scheme. Examples for objects that could be defined in this context are team members or tools that are individually characterized by their specific attribute values.

4. SESAM Simulation

Having formally described the software process, the simulation of a software project can begin. The user of the simulator (maybe a project management student) receives the project description and plans the project according to his abilities. The player then communicates with the system using a very simple interface.



```
...
AND NOW: GOOD LUCK

hire John
ok
John Bankmiller starts working. From now on you have to pay 800
DM per day of the remaining budget to John Bankmiller.

let John specify
ok
John Bankmiller says: "I really like to specify. You can talk
to many people. But sometimes they themselves don't know what
they want."

proceed 10 days
ok
The current date is 29.11.1995
...
```

Figure 4.1: The player interface of the SESAM system

Either he or she triggers actions to drive the project in the right direction, e.g. hiring new people or assigning tasks to the team members, or he/she lets time proceed giving staff members a chance to perform their tasks. In return, the player gets reactions from the system. The example shown in figure 4.1 is now discussed in some more detail. The player commands are displayed in bold-typed letters whereas the system replies are in standard type-face. The first action, the player has decided on is to "hire John". The system replies that "John Bankmiller starts working and that he costs 800 DM per day". After having hired John, the player assigns a task to him. John shall write the specification. He is delighted at this idea and says that he likes specifying very much. The third command shown differs from the previous two. Besides giving instructions and taking decisions, the player is also in charge of the simulation time. Thus he or she is able to determine the time needed for different project tasks.

As shown above, the player receives informal messages from the system depending on the actions taken. These messages mostly do not reveal any internal data. But they may give hints to the player. In the example above, the player can conclude that John Bankmiller is experienced in writing the requirements specification and that he might perform the task well. This reflects the real situation where it can be difficult for managers to get detailed knowledge of the capabilities of their staff members and of the progress of the project in general.

After having finished the simulated project, the course of the project can be analyzed. To facilitate this analysis, the SESAM system provides a so-called analysis component. This component is able to display the attribute values over time. Effects that occured during the project can be visualized and explained. Players learn which action has caused which effect. Subsequently, they can check if this effect has been a positive or a negative one with respect to the overall project. Figure 4.2 shows an example analysis of a simulated project. Assume that the progress of this project is measured in terms of recognized requirements. However, the features the developers have recognized and described in the specification document are not necessarily the same as those which the customer initially required. To deal with this problem, the player of this simulated project has decided to perform a review on the requirements specification. The curve shown in figure 4.2 reveals that the number of requirements is continually growing as long as the developers work on the specification. Then the document is given to the customer who should verify it. The customer tries to find inconsistencies and missing or obsolete functionality. As long as the customer reviews the specification, the number of requirements remains unchanged. When the customer has finished the review he reports about the findings, and the developers start to improve the document. Since the number of requirements decreases significantly, the customer has found much more obsolete than missing requirements.

The analysis component conceptually belongs to the simulation component. The modeling component and the simulation component, however, have been fully separated. This was an important design decision, since models can easily be adapted, extended, and analyzed. At the same time, the complexity of the simulation component is significantly reduced. The simulator mainly activates and deactivates the applicable rules according to the graph structure representing the state of the simulated project.

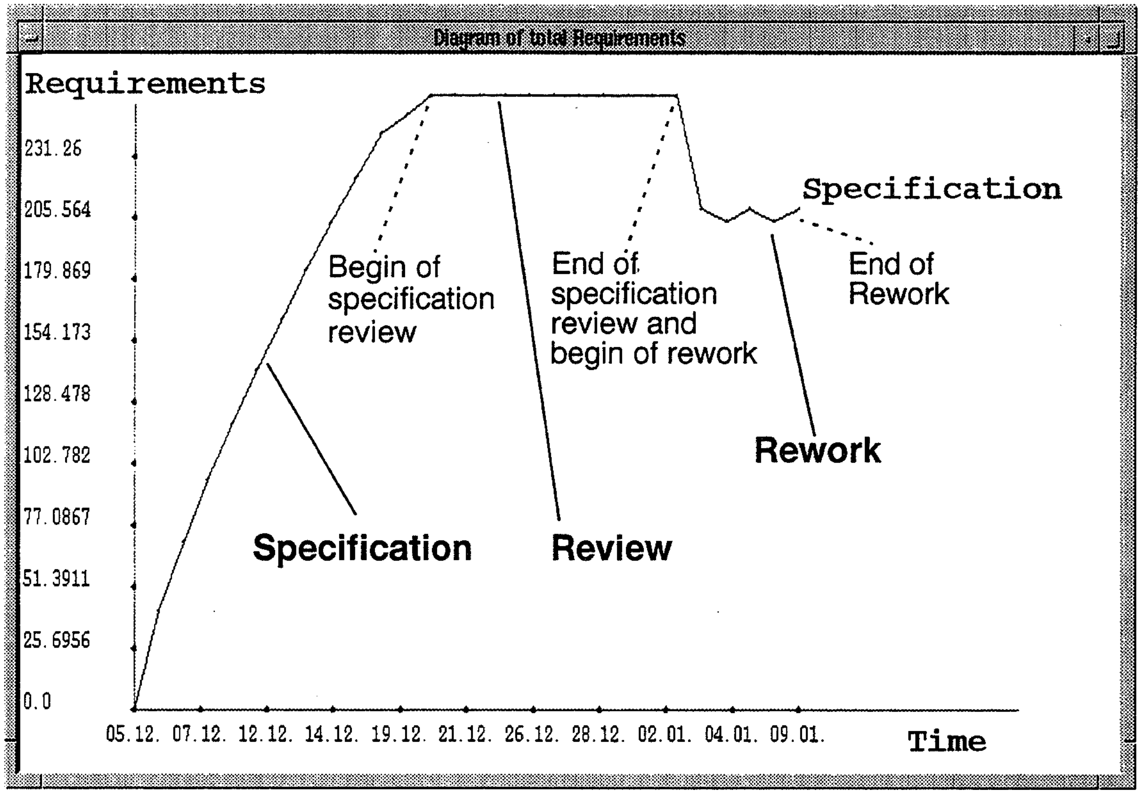


Figure 4.2: An example analysis of a simulated project

5. Experiences and Future Work

To test our modeling and simulation approach, we constructed a comprehensive model of a software process which comprises all phases of the software development lifecycle [Deininger94]. It consisted of a detailed scheme and approximately 100 rules. The model has been successfully applied in an experiment conducted at our department.

In this experiment five groups with two students each had to simulate a "check accountance" project. The experiment took twelve weeks. The students received the project information at the beginning. They had to plan and to manage the check accountance project according to their knowledge and skills. At the end, the success of the project, or its failure, respectively, was assessed using three measures: The needed time, the needed budget, and the conformance of the product with respect to the initial customer requirements. This last measure reflected how many customer requirements have been transmitted into the final product, and how many have been lost or introduced by mistake.

The results of the experiment were convincing. Students gained experiences similar to those real project managers report about. Some groups even felt panic when the simulated project got into some simulated trouble. By revealing their individual management strategies, the students gained deep insight into the software process in general and into the consequences of their individual decisions.

For research purposes however, the model proved to be less adequate. As explained above, the basic metaphor is to model product quality by a number of abstract units which initially represent customer requirements. These requirements have to be transferred from document to document to the final product. Every unit lost, or added without proper reason, decreases the quality of the emerging product. This measure appears too abstract and has no equivalent in reality, which makes it harder to validate the model. We are currently working on a new simulation model that is assembled from several modules. These modules help to reduce complexity of fine-grained models, and they can be analyzed and validated separately. The basic approach we pursue now is to collect quality aspects of software, and to identify suitable metrics to measure these quality aspects. It is important to choose metrics that can be applied to real documents in order to assure the possibility of comparing our models to real projects (the only way to validate the models). Our hope is that eventually, the models should be able to gradually replace the accumulation of rules of thumb now prevalent in the textbooks on software engineering.

References

[Abdel-Hamid91] Abdel-Hamid, T.K.; Madnick, S.E.: Software Project Dynamics: An Integrated Approach. Prentice Hall (Engelwood Cliffs), 1991.

[Carr90] Carr, D.; Koestler, R.: System Dynamics Models of Software Developments. Proceedings of the 5th International Software Process Workshop. 10. - 13. Oktober 1989, Kennebunkport, ME (USA), 1990, p. 46 - 48.

[Curtis92] Curtis, B.; Kellner, M.I.; Over, J.: Process Modeling. Communications of the ACM (35), Nr. 9, 1992, p. 75 - 90.

[Deininger94] Deininger, M.; Schneider, K.: Teaching Software Project Management by Simulation – Experiences with a Comprehensive Model. In: Díaz-Herrera, J. (Hrsg.): Software Engineering Education – 7th SEI CSEE Conference. San Antonio, Texas, USA, January 1994. Proceedings. Springer (Berlin), Lecture Notes in Computer Science No. 750, 1994, p. 227 - 242.

[Drappa95] Drappa, A.; Melchisedech, R.: The Use of Graph Grammar in a Software Engineering Education Tool. Proceedings of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation. 28. August - 01. September 1995, Volterra (Italien), 1995, p. 133 - 140.

[Finkelstein94] Finkelstein, A.; Kramer, J.; Nuseibeh, B. (Eds.): Software Process Modelling and Technology. John Wiley & Sons Inc. (New York), 1994.

[Levary91] Levary, R.R.; Lin, C.Y.: Modelling the Software Development Process using an Expert Simulation System Having Fuzzy Logic. Software - Practice and Experience (21/2), 1991, p. 133 - 148.

[Ludewig89] Ludewig, J.: Modelle der Software-Entwicklung – Abbilder oder Vorbilder? Softwaretechnik-Trends (9), No. 3, 1989, p. 1 - 12 (in german).

[Ludewig92] Ludewig, J.; Bassler, T.; Deininger, M.; Schneider, K.; Schwille, J.: SESAM – Simulating Software Projects. Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering. Juni, 1992, Capri (Italien), 1992, p. 608 - 615.

[Schneider94] Schneider, K.: Ausführbare Modelle der Software-Entwicklung. vdf Hochschulverlag (Zürich), 1994 (in german).

[Snowdon94] Snowdon, R.A.; Warboys, B.C.: An Introduction to Process-Centred Environments. In: Finkelstein, A.; Kramer, J.; Nuseibeh, B. (Hrsg.): Software Process Modelling and Technology. John Wiley & Sons Inc. (New York), 1994, p. 1 - 8.

Modeling and Simulation of Software Projects with SESAM

Anke Drappa
University of Stuttgart
drappa@informatik.uni-stuttgart.de

Overview

Part I: The Problem – Software Engineering Education
Part II: The Solution – The SESAM System
Part III: The Results – Experiences with the System

A Tough Problem in Software Engineering

- Teaching project managers adequately seems hard
- Current approaches:
  – learning from textbooks
  – learning by leading real projects

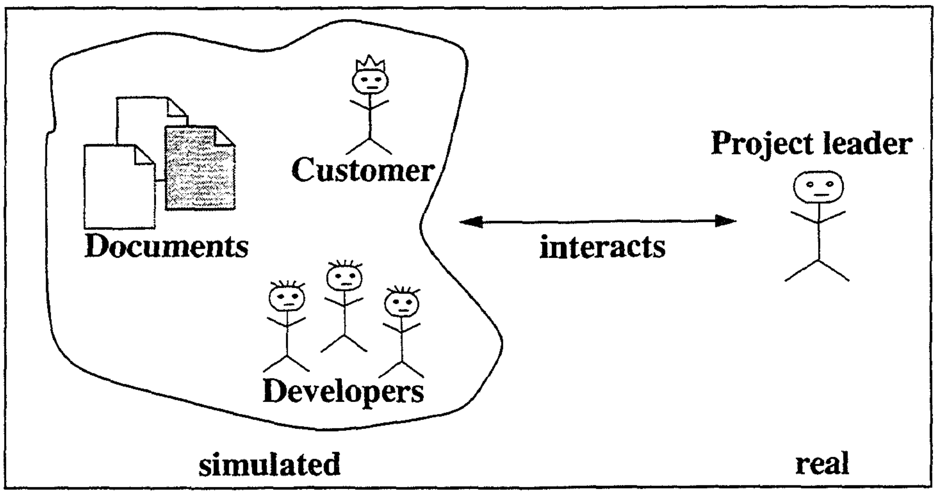
A Solution from an other Area

- Simulation
- Example:
  – teaching pilots using a flight simulator
- Is that something we can apply to Software Engineering?

Part I: **The Problem**

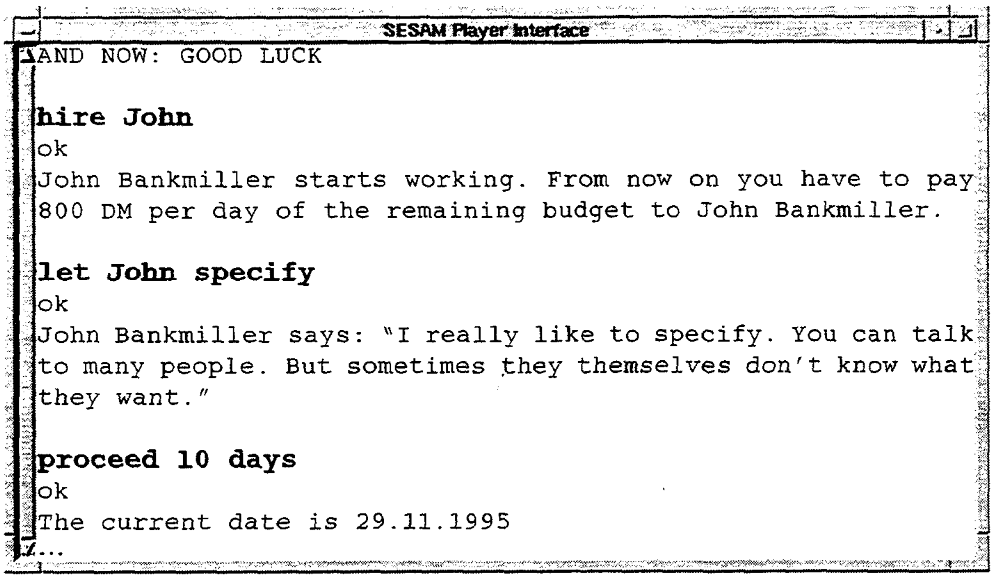
Research Project SESAM:

- Software Engineering Simulation by Animated Models



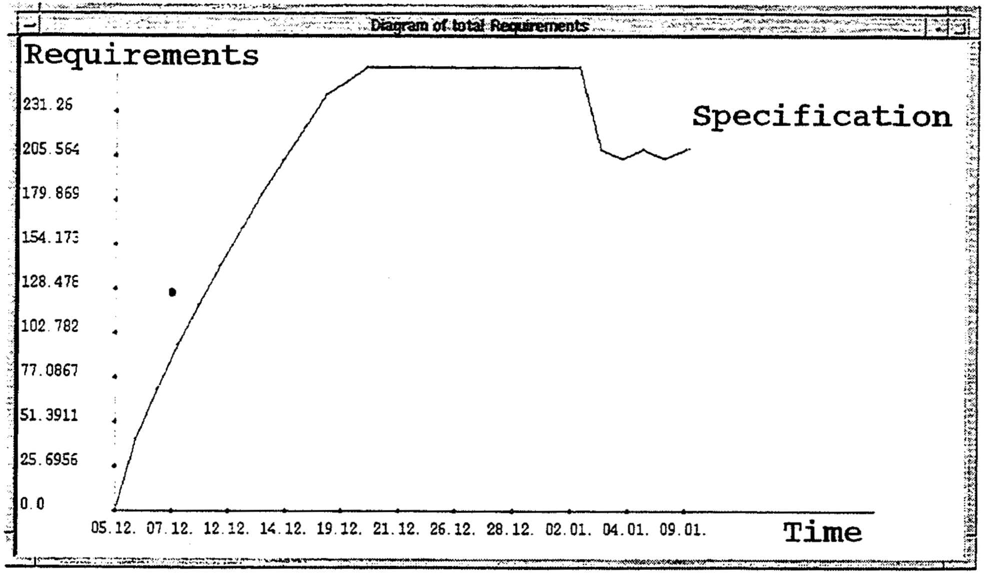
Part II: **The Solution – SESAM System**

The Player´s Interface of the SESAM System

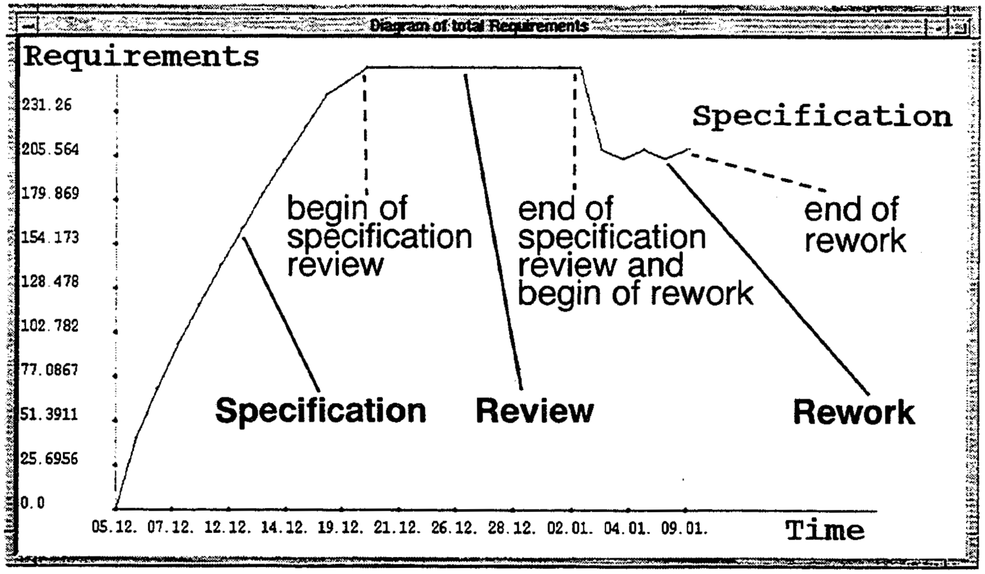


```
AND NOW: GOOD LUCK

hire John
ok
John Bankmiller starts working. From now on you have to pay
800 DM per day of the remaining budget to John Bankmiller.

let John specify
ok
John Bankmiller says: "I really like to specify. You can talk
to many people. But sometimes they themselves don't know what
they want."

proceed 10 days
ok
The current date is 29.11.1995
...
```

The Analysis Component of the SESAM System



The Analysis Component of the SESAM System



begin of specification review

end of specification review and begin of rework

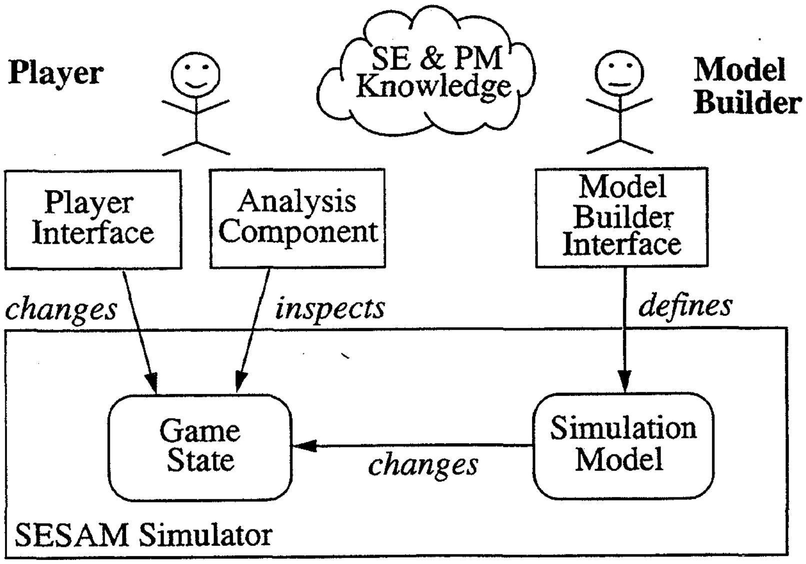
end of rework

Specification

Review

Rework

Architecture of the SESAM System



Model Building in SESAM: Basic Ideas

- Modeling simple effects where each effect influences the course of the simulated project
- Building fine-grained models allowing the player to gain reality-like experiences
- Building descriptive models

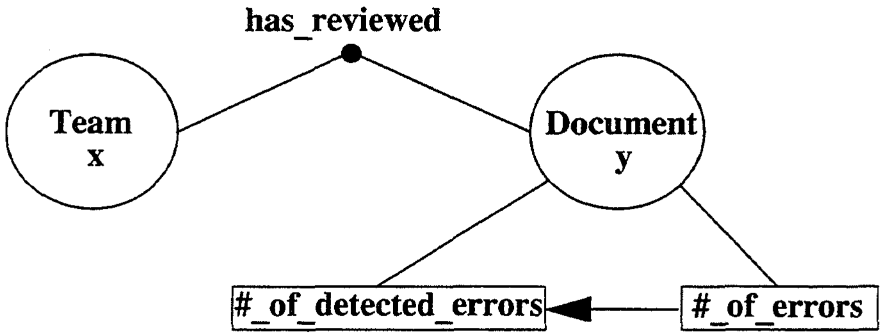
An Example

"Walkthroughs catch 60 percent of the errors."

Boehm, B.: Industrial Software Metrics Top 10 List. IEEE Software (No. 5), 1987, p. 84-85.

Part III: **The Solution – SESAM System**

Model Building in SESAM: Formal Description



Equation: **y.#_of_detected_errors := y.#_of_errors * 0.6**

Part III: **The Results**

Simulating Software Projects: An Experiment

- 5 groups with 2 students each simulated a "check accountance" project
- we used a simple model which covered all phases of software development
- success of the simulated projects was measured considering recognized requirements, time, and budget

➪ the students gained experiences similar to those real project managers report about

➪ "post-mortem" analysis of the project allowed deep insight into the software development process

Future Work

- Building models which are
  - more detailed,
  - more realistic, and
  - cover more aspects of the software development process
- Bolstering models by quantitative data which is collected in experiments with students conducted in our department
- Enhancing functionality of the SESAM simulation system

SE
SAM

http://www.informatik.uni-stuttgart.de/ifi/se/projekte/Sesam.html

Evaluating Empirical Models for the Detection of High-Risk Components: Some Lessons Learned

Filippo Lanubile[1] and Giuseppe Visaggio

Dipartimento di Informatica, University of Bari
Via Orabona 4, 70126 Bari, Italy
email: [lanubile|visaggio]@seldi.uniba.it

513-61
415609
360703
16 P.

1. Introduction

Software complexity metrics are often used as indirect metrics of reliability since they can be obtained relatively early in the software development life cycle. Using complexity metrics to identify high-risk components, i.e. components which likely contain faults, allows software engineers to focus the verification effort on them, thus achieving a reliable product at a lower cost. Since in this study the direct metric of reliability is the class to which the software component belongs (high-risk or low-risk), the prediction problem is reduced to a classification model.

Classification problems have traditionally been solved by various methods, which originate from different problem-solving paradigms such as statistical analysis, machine learning, and neural networks. This study compares different modeling techniques which cover all the three classification paradigms: principal component analysis, discriminant analysis, logistic regression, logical classification models, layered neural networks, and holographic networks. A detailed description of our implementation choices in building the classification models can be found in [LLV95].

2. Data Description

Raw data were obtained from 27 projects performed in a software engineering course at the University of Bari, by different three student-teams over a period of 4-10 months. The systems, business applications developed from a same specification, range in size from 1100 to 9400 lines of Pascal source code. From each system, we randomly selected a group of 4-5 components for a total of 118 components, ranging in size from 60 to 530 lines of code. Here, the term software component refers to functional abstractions of code such as procedures, functions and main programs. Each group of component was tested by independent student teams of an advanced software engineering course with the aim to find faults.

In order to build unbiased classification models, we decided to have an approximately equal number of components in the classes of reliability. Thus, we defined as high-risk any software component where faults were detected during testing, and low-risk any component with no faults discovered.

[1] Filippo Lanubile is spending a sabbatical period at the University of Maryland, College Park.

In addition to the fault data, 11 software complexity metrics were used to construct the classification models:

1. McCabe's cyclomatic complexity ($v(G)$)
2. Halstead's number of unique operands (η_2)
3. Halstead's total number of operands (N_2)
4. Total number of lines of code (*LOC*)
5. Number of non-comment lines of code (*NCLOC*)
6. Halstead's program length (*N*)
7. Halstead's volume (*V*)
8. Henry&Kafura's fan-in (*fanin*)
9. Henry&Kafura's fan-out (*fanout*)
10. Henry&Kafura's information flow (*IF*)
11. density of comments (*DC*)

The metrics have been selected so as to measure both design and implementation attributes of the components, such as control flow structure (metric 1), data structure (metrics 2-3), size (metrics 4-7), coupling (metrics 8-10), and documentation (metric 11). Most of these metrics have been already used in other empirical studies to test predictive models with respect to faults [MK92, LK94], and program changes [KLM93, KS94, LK94].

The set of 118 observations was subsequently divided into two groups. Two thirds of the components, made up of 79 observations, were randomly selected to create and tune the predictive models. The remaining 39 observations provided the data to test the models and compare their performances. From now on, the first group of observations will be called training set, while the second one testing set.

3. Evaluation Criteria

We selected statistical criteria which are based on the analysis of categorical data. In our study we have two variables (*real risk* and *predicted risk*) that can assume only two discrete values (*low* and *high*) in a nominal scale. Then the data can be represented by a two-dimensional contingency table with one row for each level of the variable *real risk* and one column for each level of the variable *predicted risk*. The evaluation criteria are predictive validity, misclassification rate, achieved quality and verification cost.

The predictive validity is the capability of the model to predict the future component behavior from present and past behavior. The present and past behavior is represented by data in the training set while the future behavior of components is described by data in the testing set. In our context, where data are represented by a contingency table, we apply the predictive validity by testing the null hypothesis of no association between the row variable (*real risk*) and the column variable (*predicted risk*). In this case, the predictive model is not able to discriminate low-risk components from high-risk components. The alternative hypothesis is one of general association. A chi-square (χ^2) statistic with a distribution of one degree of freedom is applied to test the null hypothesis.

We use the criterion of predictive validity for assessment, since we determine the absolute worth of a predictive model by looking at its statistical significance. A model which does not meet the criterion of predictive validity should be rejected. The remaining criteria are used for comparison, taking into account that the choice between the accepted models depends from the perspective of the software engineering manager. In practice he could be more interested in achieving a better quality at a high verification cost or be satisfied of a lower quality, sparing verification effort.

For our predictive models, which classify components as either low-risk or high-risk, two misclassification errors are possible. A Type 1 error is made when a high-risk component is classified as low-risk, while a Type 2 when a low-risk component is classified as high-risk. It is desirable to have both types of error small. However, since the two types of errors are not independent, software engineering managers should consider their different implications. As a result of Type 1 error, a component actually being high-risk could pass the quality control. This would cause the release of a lower quality product and more fix effort when a failure will happen. As a result of Type 2 error, a component actually being low-risk will receive more testing and inspection effort than needed. This would cause a waste of effort. We adopt from [Sch94], as measures of misclassification, the proportions of Type 1, Type 2, and Type 1 + Type 2 errors.

We are interested in measuring how effective are the predictive models in terms of the quality achieved after that the components classified as high-risk have been undergone to a verification activity. We suppose that the verification will be so exhaustive to find the faults of all the components which are actually high-risk. We measure this criterion using the completeness measure [BTH93], which is the percentage of faulty components that have been actually classified as such by the model.

Quality is achieved by increasing the cost of verification due to an extra effort in inspection and testing for the components which have been flagged as high-risk. We measure the verification cost by using two indicators. The former, inspection [Sch94], measures the overall cost by considering the percentage of components which should be verified. The latter, wasted inspection, is the percentage of verified components which do not contain faults because they have been incorrectly classified.

4. Analysis of the Results

We applied the evaluation criteria on the testing set and analyzed the resulting data, shown in * Table 1. The first two columns show the chi-square values and the significance levels across the classification models built using different modeling techniques. From the low values of chi-square and high significance levels in the testing set, we accept the null hypothesis of no association between predicted risk and real risk. In fact, all the values of significance are too high with respect to the most common values which are used to reject the null hypothesis.

As regards the misclassification rate, we recall that a casual prediction should have 50 percent of proportion of Type 1 + Type 2, and 25 percent for both proportion of Type 1 and Type 2. From data showing the misclassification rates, we see that the proportion of Type 1 + Type 2 error ranges between 46 percent and 59 percent. Discriminant analysis and logistic regression, when applied in conjunction with principal component analysis, have a high proportion of Type 2 error (respectively 41 and 46 percent) with respect to the proportion of Type 1 error

α is the probability of uncorrectly rejecting the null hypothesis (no association) (respectively 15 and 13 percent). On the contrary, the other models have balanced values of Type 1 and Type 2 error, ranging between 20 and 28 percent.

	Predictive Validity		Misclassification Rate			Achvd Quality	Verification Cost	
Modeling Techniques	χ^2	α^*	P_1	P_2	P_{12}	C	I	WI
Discriminant anal.	0.244	0.621	28.21	25.64	53.85	42.11	46.15	55.56
Discriminant anal. + Principal cmpnt.	0.685	0.408	15.38	41.03	56.41	68.42	74.36	55.17
Logistic regression	0.648	0.421	28.21	28.21	56.41	42.11	48.72	57.89
Logistic regression + Principal cmpnt	1.761	0.184	12.82	46.15	58.97	73.68	82.05	56.25
Logical classif. model	0.215	0.643	25.64	20.51	46.15	47.37	43.59	47.06
Layered neural ntwrk	0.648	0.421	28.21	28.21	56.41	42.11	48.72	57.89
Holographic ntwrk	0.227	0.634	25.64	28.21	53.85	47.37	51.28	55.00

* α is the probability of uncorrectly rejecting the null hypothesis (no association)

Table 1. Results of evaluation criteria

Looking at the achieved quality and the verification cost, we can interpret better the misclassification results. In fact, the highest values of quality correspond to the models built with principal component analysis followed from either discriminant analysis or logistic regression (completeness is, respectively, 68 and 74 percent). But these high values of quality are obtained by inspecting the great majority of components (inspection is, respectively, 74 and 82 percent), thus wasting more than one half of the verification effort (wasted inspection is, respectively, 55 and 56 percent). The lowest level of quality (completeness is 42 percent) is achieved by both discriminant analysis and logistic regression, when used without principal component analysis, and by the layered neural network. The logistic regression without principal components and the layered neural network have also the poorest results in correctly identifying the high-risk components (wasted inspection is 58 percent). On the contrary the logical classification model is the only model which dissipates less than one half of the verification effort (wasted inspection is 47 percent).

5. Lessons learned

This empirical investigation of the modeling techniques for identifying high-risk modules has taught us three lessons:

- Predicting the future behavior of software products does not always lead to successful results. Despite of the variegated selection of modeling techniques, no model satisfies the criterion of predictive validity, that is no model is able to discriminate between components with faults and components without faults. This result is in contrast with various papers which report successful results in recognizing fault-prone components from analogous sets of complexity measures.

Briand *et al.* [BBH93] presented an experiment for predicting high-risk components using two logical classification models (Optimized Set Reduction and classification tree) and two logistic regression models (with and without principal components). Design and code metrics were collected from 146 components of a 260 KLOC system. OSR classifications were found to be the most complete (96 percent) and correct (92 percent), where correctness is the complement of our wasted inspection. The classification tree was more complete (82 percent) and correct (83 percent) than logistic regression models. The use of principal components improved the accuracy of logistic regression, from 67 to 71 percent of completeness and from 77 to 80 percent of correctness.

Porter [Por93] presented an application of classification trees to data collected from 1400 components of six FORTRAN projects in NASA environment. For each component, 19 attributes were measured, capturing information spanning from design specifications to implementation. He measured the mean accuracy across all tree applications according to completeness (82 percent) and to the percentage of components whose target class membership is correctly identified (72 percent), that is the complement of the Proportion of Type 1 and Type 2 error.

Munson and Koshgoftaar [MK92] detected faulty components by applying principal component analysis and discriminant analysis to discriminate between programs with less than five faults and programs having 5 or more faults. The data set included 327 program modules from two distinct Ada projects of a command and control communication system. They collected 14 metrics, including Halstead's metrics together with other code metrics. Applying discriminant analysis with principal components, at a probability level of 80 percent, resulted in recognizing 79 percent of the modules with a total misclassification rate of 5 percent.

Our result is closer with the investigation performed by Basili and Perricone [BP84], where the unexpected result was that module size and cyclomatic complexity had no relationship with the number of faults, although there was a negative relationship with the fault density.

- Predictive modeling techniques are only as good as the data they are based on. The relationship between software complexity measures and software faults cannot be considered an assumption which holds for any data set and project. A predictive model, from the simplest to the most complex, is worthwhile only if there is a local process to select metrics which are valid as predictors.

- Principal component analysis does not always produce a better input for predictive models. The domain metrics have often been used in the software engineering field [BBH93, BTH93, MK92, KLM93] to reduce the dimensions of a metric space when the metrics have a strong relationship between them, and obtain a smaller number of orthogonal domain metrics to be used as input to regression and discriminant analysis models. In our study, we built two classification models for both discriminant analysis and logistic regression. The first couple of models was based on the eleven original complexity measures, while the second one used the three domain metrics which had been generated from the principal component analysis. An unexpected result of the models using orthogonal domain metrics is that the good performance in achieved quality is exclusively the result of classifying very often components to be high-risk.

Acknowledgments

Our thanks to Aurora Lonigro and Giulia Festino for their help in the preparation, execution and analysis of this study.

References

[BTH93] L. C. Briand, W. M. Thomas, and C. J. Hetmanski, "Modeling and managing risk early in software development", in *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993, pp.55-65.

[BBH93] L. C. Briand, V. R. Basili, and C. J. Hetmanski, "Developing interpretable models with optimized set reduction for identifying high-risk software components", *IEEE Transactions on Software Engineering*, vol.19, no.11, November 1993, pp.1028-1044.

[KLM93] T. M. Khoshgoftaar, D. L. Lanning, and J. C. Munson, "A comparative study of predictive models for program changes during system testing and maintenance", in *Proceedings of the Conference on Software Maintenance*, Montreal, Canada, September 1993, pp.72-79.

[KS94] T. M. Khoshgoftaar, and R. M. Szabo, "Improving code churn prediction during the system test and maintenance phases", in *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 1994, pp.58-67.

[LK94] D. L. Lanning, and T. M. Khoshgoftaar "Canonical modeling of software complexity and fault correction activity", in *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 1994, pp.374-381.

[LLV95] F. Lanubile, A. Lonigro, and G. Visaggio, "Comparing models for identifying fault-prone software components", in *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, Rockville, Maryland, USA, June 1995, pp.312-319.

[MK92] J. C. Munson, and T. M. Khoshgoftaar, "The detection of fault-prone programs", *IEEE Transactions on Software Engineering*, vol.18, no.5, May 1992, pp.423-433.

[Por93] A. A. Porter, "Developing and analyzing classification rules for predicting faulty software components", in *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, June 1993, pp.453-461.

[Qui93] J. R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kauffman Publishers, San Mateo, CA, 1993.

[Sch94] N. F. Schneidewind, "Validating metrics for ensuring Space Shuttle Flight software quality", *Computer*, August 1994, pp.50-57.

20th Annual
Software Engineering Workshop

Evaluating Empirical Models for the Detection of High-Risk Components: Some Lessons Learned

Filippo Lanubile   Giuseppe Visaggio
University of Bari, Italy

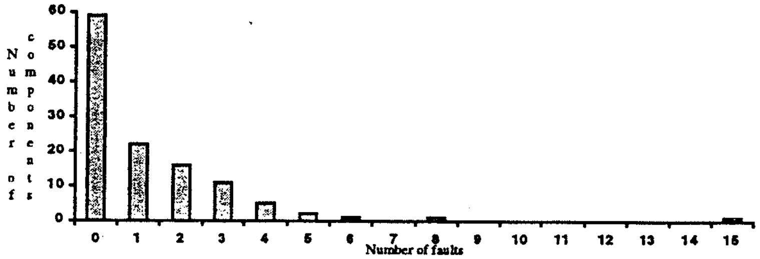
Research Questions

- Are the modeling techniques (from statistical analysis, machine learning, and neural networks) useful to predict the reliability of software components?
- What modeling techniques perform better?
- Under which conditions?
  - Predictive model   R = Fi (R1, R2, ..., Rn)
    - » R1, R2, ..., Rn are indirect measures of reliability
    - » R is a direct measure of reliability
    - » Fi   is a modeling technique

Direct Metric

- Risk-class to which the software component belongs
  - high-risk: any software component with faults detected during testing
  - low-risk: any software component with no faults detected

The prediction model is reduced to a classification model



Indirect Metrics

- Control flow: cyclomatic complexity (v(G))
- Data structure: number of unique operands (n2), total number of operands (N2)
- Size: number of lines of code (LOC), number of non-comment lines of code (NCLOC), program length (N), volume (V)
- Coupling: fanin, fanout, information flow (IF).
- Documentation: density of comments (DC)

Modeling Techniques

- Statistical analysis
  - Discriminant Analysis
  - Principal Component Analysis + Discriminant Analysis
  - Logistic Regression
  - Principal Component Analysis + Logistic Regression
- Machine Learning
  - Logical Classification Models
- Neural Network
  - Layered Neural Networks
  - Holographic Networks

Environment

- Software engineering course with 27 information system projects
  - same specification but developed by different three person-teams
  - moderate-sized Pascal programs (1-9 KLOC)

Data

- Random selection of 118 components (4-5 components from each program)
- Unit tests performed by independent student teams on the 118 components
- Random division of data between
  - Model creation
    - » training set: observations from 2/3 of components
  - Model evaluation
    - » testing set: observations from 1/3 of components

Evaluation Criteria

	Predicted Risk		
Real Risk	***low***	***high***	
low	n_{11}	n_{12}	$n_{1\bullet}$
high	n_{21}	n_{22}	$n_{2\bullet}$
	$n_{\bullet 1}$	$n_{\bullet 2}$	n

- Preditive validity

- Misclassification rate
- Quality achieved
- Verification cost

Predictive Validity

	Predicted Risk		
Real Risk	***low***	***high***	
low	n_{11}	n_{12}	$n_{1\bullet}$
high	n_{21}	n_{22}	$n_{2\bullet}$
	$n_{\bullet 1}$	$n_{\bullet 2}$	n

- Capability of the model to predict the future component behavior (testing set) from present and past behavior (training set)
  - Null hypothesis: no association between the row variable (real risk) and the column variable (predicted risk)

Evaluation of Predictive Validity

Modeling Techniques	χ^2	p
Discriminant anal.	0.244	0.621
Discriminant anal.+Principal cmpnts	0.685	0.408
Logistic regression	0.648	0.421
Logistic regression+Principal cmpnts	1.761	0.184
Logical classification model	0.215	0.643
Layered neural network	0.648	0.421
Holographic network	0.227	0.634

if $p > 0.05$ there is no significant association

Misclassification rate

	Predicted Risk		
Real Risk	***low***	***high***	
low	n_{11}	n_{12} **Type 2**	$n_{1\bullet}$
high	n_{21} **Type 1**	n_{22}	$n_{2\bullet}$
	$n_{\bullet 1}$	$n_{\bullet 2}$	n

- Proportion of Type 1: P1 = n_{21} / n
- Proportion of Type 2: P2 = n_{12} / n
- Proportion of Type 1+Type 2: P_{12} = (n_{21} + n_{12}) / n

Evaluation of Misclassification Rate

Modeling Techniques	P_1	P_2	P_{12}
Discriminant anal.	28.21	25.64	53.85
Discriminant anal.+Princ.cmpmts	15.38	41.03	56.41
Logistic regression	28.21	28.21	56.41
Logistic regression+Princ.cmpnts	12.82	46.15	58.97
Logical classification model	25.64	20.51	46.15
Layered neural network	28.21	28.21	56.41
Holographic network	25.64	28.21	53.85

Achieved Quality

	Predicted Risk		
Real Risk	*low*	*high*	
low	n_{11}	n_{12}	$n_{1\bullet}$
high	n_{21}	n_{22}	$n_{2\bullet}$
	$n_{\bullet 1}$	$n_{\bullet 2}$	n

How effective are the predictive models in terms of the quality achieved after that the components classified as high-risk have been undergone to a verification activity

- Completeness: $C = n_{22} / n_{2\bullet}$

Evaluation of Achieved Quality

Modeling Techniques	C
Discriminant anal.	42.11
Discriminant anal.+ Princ. cmpnts	68.42
Logistic regression	42.11
Logistic regression+Princ. cmpnts	73.68
Logical classification model	47.37
Layered neural network	42.11
Holographic network	47.37

Verification Cost

	Predicted Risk		
Real Risk	***low***	***high***	
low	n_{11}	n_{12}	$n_{1\bullet}$
high	n_{21}	n_{22}	$n_{2\bullet}$
	$n_{\bullet 1}$	$n_{\bullet 2}$	n

Extra effort in inspection and testing for the components which have been flagged as high-risk

- Inspection: $I = n_{\bullet 2} / n$
- Wasted Inspection: $WI = n_{12} / n_{\bullet 2}$

Evaluation of Verification Cost

Modeling Techniques	I	WI
Discriminant anal.	46.15	55.56
Discriminant anal.+ Princ. cmpnts	74.36	55.17
Logistic regression	48.72	57.89
Logistic regression+Princ. cmpnts	82.05	56.25
Logical classification model	43.59	47.06
Layered neural network	48.72	57.89
Holographic network	51.28	55.00

Lessons learned - 1

- Predicting the future behavior of software products does not always lead to successful results
  - no model satisfies the criterion of predictive validity
  - contrast with various studies reporting successful results in recognizing fault-prone components from analogous sets of complexity measures

Lessons learned - 2

- Predictive modeling techniques are only as good as the data they are based on
  - The relationship between software complexity measures and software faults cannot be considered an assumption which holds for any data set and project
  - A predictive model is worthwhile only if there is a local process to select metrics which are valid as predictors

Lessons learned - 3

- Principal component analysis does not always produce a better input for predictive models
  - In our study, the models using orthogonal domain metrics show better performance in achieved quality as a result of classifying very often components to be high-risk

Omit
[illegible]

[illegible]

Session 5: Method Evaluation

Object-Oriented Software Metrics for Predicting Reusability and Estimating Size
Pete Sanderson, Southwest Missouri State University

Improving the Software Testing Process in NASA's Software Engineering Laboratory
Sharon Waligora, Computer Sciences Corporation

How Do Formal Methods Affect Code Quality?
Shari Lawrence Pfleeger, Systems/Software, Inc.

OBJECT-ORIENTED SOFTWARE METRICS FOR PREDICTING REUSABILITY AND ESTIMATING SIZE

D. Peter Sanderson[*]
Department of Computer Science
Southwest Missouri State University
Springfield, MO 65804
dps910f@cnas.smsu.edu

Tuyet-Lan Tran, Josef S. Sherif, Susan S. Lee
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

ABSTRACT

As object-oriented software development methods come into more widespread use, basic questions of software quality assurance must be reconsidered. We will highlight efforts now underway at NASA's Jet Propulsion Laboratory to both assess the quality of software systems developed using object-oriented technology and develop guidelines for future development of such systems. The current focus is on design and code reusability, and system size estimation. A number of metrics are proposed, and two JPL software systems measured and analyzed. The preliminary results reported here should be particularly useful to software development and quality assurance personnel working in a C++ implementation environment.

1. INTRODUCTION

The concepts and use of software metrics in quality assurance are well-established. However, traditional metrics were developed and validated for a design methodology in which system functional and data elements are distinguished. The object-oriented methodology combines them, necessitating reconsideration of traditional metrics and motivating the quest for new ones. Metrics are best applied as part of a quality assurance strategy that considers both the purpose for measuring (such as estimation, prediction, assessment, and improvement) and characteristics of quality (such as reliability, reusability, maintainability, and readability).

The current focus of our work is twofold: predicting design and code reusability and estimating software size. Both contribute to economical software production. However, metrics selected and adapted pursuant to these goals may be applied to others as well. For instance, size metrics predict maintenance as well as development complexity and costs. A metric may also contribute toward contradictory quality goals. For instance, the inheritance property allows an object of one class to possess components and perform operations defined in a related class. This contributes both toward greater reuse, a positive factor, and greater coupling between classes, a negative one.

2. BACKGROUND

The object-oriented technologies are based on the concept of a software system as a collection of interacting objects. Objects encapsulate both the state and behavior of identifiable abstractions in the application domain. The class is a mechanism for specifying object types. The class declares both data and function members. The data members, also called instance variables, specify object state and are typically private to the class. The member functions, also called instance methods (or just methods), specify object behaviors and are typically accessible

[**] Supported in part by a 1995 NASA/ASEE Summer Faculty Fellowship.

throughout the system. Given the identity of an object, a client object may interact with it by sending it a message, which invokes the corresponding method. It is through such interactions that computation proceeds in object-oriented systems. Classes may be related to other classes through specialization, in which a class is defined to inherit the attributes of another, more general, class and extend it through additional, unique characteristics. Such inheritance provides a measure of reuse, since the child class possesses the variables and methods of its parent class without explicitly defining them.

A large number of software metrics have been developed based on the procedural paradigm. As a result, their focus is on measuring the characteristics of procedures. Examples are Halstead's software science [Halstead 77] which measures complexity based on operator and operand counts, and McCabe's cyclomatic complexity [McCabe 76] which measures complexity based on the number of control paths in a procedure. The measurement of object-oriented software, however, should focus the major elements of the object-oriented paradigm: classes and objects.

The seminal work in this area of that of Chidamber and Kemerer at MIT, as most recently reported in [Chidamber and Kemerer 94]. They present a suite of six metrics designed to measure complexity in the design of classes. The theoretical basis for each metric is explained, and measurement results from two software development sites is summarized. The versatility of these metrics is demonstrated by their incorporation into other metric programs, such as those reported by Li and Henry 93, Lorenz and Kidd 94, Rosenberg 95]. They are utilized in our work as well, and will be cited below as appropriate. A large metric suite including metrics for size and reuse is reported in [Abreu and Carapuca 94].

3. THE METRICS

We have defined two small metrics suites, one to assess system size and the other reuse and reusability. They are by no means exhaustive, but contain metrics relevant to their intended purpose. The metrics are quantifiable, and easily collected from detailed design documents or source code files. Thus they measure static system qualities. We were able to quickly develop a software tool to collect most of the measurements from C++ source files. The metrics for both suites are described here, with thresholds for all metrics described in the analysis of the two JPL systems measured.

3.1 System Size Metrics

We measure system size using three metrics for comparison purposes. All are based on the sum of the class sizes. In a completely object-oriented system, all system functionality is contained within class method definitions. In hybrid languages such as C++, which allow non-object-oriented structures, this is rarely the case. If a substantial portion of an application is known to be non-object-oriented, the system size metric should be supplemented with measurements from conventional size metrics.

If each class is assigned a size of unity, the metric becomes **number of classes**. Alternatively, each class can be sized according to the sum of its method sizes. Only those methods defined in a class are included, to assure that inherited methods are not counted repeatedly. If each method is assigned a size of unity, the metric becomes **number of methods**. Alternatively, each method can be sized using any of a number of available metrics such as the McCabe or Halstead measures. Since such measures assign weights to individual methods, the resulting system size metric is the **sum of method weights**. We have selected **non-comment source statements** (NCSS) as the basis for weighing methods [Grady 92]. This differs from lines of code in that comments are not included, and that the free-format syntax of modern programming languages is taken into account.

3.2 Reuse and Reusability Metrics

There are different forms of design and code reuse; we focus on that which naturally results from method inheritance based on class hierarchies. Such hierarchies are formed when subclasses are defined as specializations of other classes. A subclass inherits the variables and methods of the more general class from which it is derived. Indeed, reuse increases as the inheritance tree of classes changes through iterative system development. As mentioned above, this also contributes to a higher degree of coupling between the related classes. Reuse through inheritance is measured using two of the Chidamber and Kemerer metrics, **number of child classes** (NOC) and **depth in inheritance tree** (DIT). Both metrics are based on the inheritance hierarchy structures, which are trees since we consider single inheritance only (a class may have but one parent). NOC is the number of classes directly derived from the subject class. DIT is the number of hops required to reach the root of the inheritance tree in which the subject class resides.

Reusability metrics are those which attempt to assess the potential for reuse of existing design or code. This is more difficult to measure than system size or reuse, yet is potentially the most significant in terms of development cost savings. Metrics contributing toward an assessment of reusability include **NOC** and **DIT** from above, plus **coupling between object classes** (CBO) [Chidamber and Kemerer 94], the **number of instance variables** per class and **number of methods** per class. The CBO metric determines coupling by the number of messages objects send to each other. This occurs whenever a method is invoked via an object. An instance variable is a class data member which is instantiated for each new object. Analysis of the distributions of these metrics is as useful as that of the averages and extremes.

4. THE MEASUREMENTS

We measured two existing C++ software systems with respect to the metrics and goals described above. Both are JPL applications developed to serve specific spacecraft communication and control support functions: (1) The Sequence Generator (SEQ_GEN), which is an element in the sequence subsystem of the Advanced Multimission Operations System, and (2) The Microwave Generic Controller (UGC) for the 34 meter Beam Waveguide Antenna. Our UGC system focus is on the code comprising the Generic Kernel (UGC_GK) software component. Neither system is completely object-oriented; size metrics for functions not associated with a class were collected but are not reported here.

Metric values for the SEQ_GEN and UGC_GK source code were collected using a measurement tool developed by the principal author during a summer fellowship at JPL [Sanderson 95]. The tool is a collection of UNIX shell scripts, AWK scripts and C programs which communicate through the shell's pipe and redirection capabilities. All the above described metrics except coupling between objects (CBO) were measured. Measurements of metrics for system size estimation are summarized in Tables 1 through 3. Measurements for reuse and reusability assessment are summarized in Figures 1 through 5.

5. ANALYSIS

Measurements from SEQ_GEN and UGC_GK were analyzed, and the results reported. Since the applications are unequal size, charted results have been normalized to allow direct comparison. When explanations cite other results in the literature, they refer to the charts presented in [Chidamber and Kemerer 94] and [Lorenz and Kidd 94].

5.1 System Size

Table 1 contains the system size measurements for the SEQ_GEN and UGC_GK applications. The last column indicates the relative size of SEQ_GEN to UGC_GK under the three metrics.
Size ratios based on class and method counts are nearly identical, since the average number of methods per class are about the same for the two applications (see Table 2). If system size is measured by the sum of method weights, SEQ_GEN is seen as relatively much smaller, about four times the size of UGC_GK rather than about seven times. There is currently no consensus in the literature on which metric is best.

5.2 Reuse

The more interesting metrics are those to assess reuse and potential for reuse. We first consider evidence of reuse through inheritance. All measurements were taken from the production versions of SEQ_GEN and UGC_GK, thus it was not possible to track reuse as system development progressed through several preliminary versions.

Number of Child Classes: Distributions for both systems are shown in Figure 1, normalized to the percent of classes for each value for easy comparison. The distributions appear quite similar; both indicate that very few classes benefit from inheritance and thus code reuse is low. All classes with NOC value 0 represent the "leaf" nodes in the tree formed by the system class hierarchy. The proportion of such classes is expected to be high in any case (over 50 percent for a perfect binary tree, for example). Classes having high NOC values, however, may warrant individual attention. For example, one SEQ_GEN class has 56 child classes.

Typically only one class (often named "Object" or "Node") will possess such an attribute. Metric values for individual classes are included on the report (not shown here) generated by the measurement tool.

Depth In Tree: DIT distributions are summarized in Figure 2. In this case the maximum values are similar but the distributions quite different (just the opposite of NOC). A class having DIT value of 0 is by definition the root of a class hierarchy. Since nearly two-thirds of UGC_GK classes are therefore roots and over 80% of them are leaves, the application consists mainly of single unrelated classes. This indicates a low level of reuse through inheritance, due to either the nature or size of the application. The SEQ_GEN metric is more normally distributed, and is more characteristic of systems analyzed in the literature. The application provides many related abstractions, and the designers were able to exploit them. In any case, the maximum DIT should rarely exceed seven; such subclasses may not really be specializations of the classes from which they inherit.

5.3 Reusability

Most of the metrics with which we are experimenting are directed toward assessing the potential reusability of existing designs and code. As these efforts are still in the early stages, the number of candidate metrics is subject to revision and refinement. We expect candidates to be refined as research progresses until a small but significant set remains. The analysis of SEQ_GEN and UGC_GK considers the averages, maxima, and distributions of the metrics.

Number of Variables: The number of instance variables indicates object size and complexity. Each instance variable is declared as a data member. The measurement tool currently counts only the variables explicitly declared in a class definition; it will be enhanced to also include inherited variables. The averages and distributions are given in Table 2 and Figure 3, respectively. The distribution of values across the SEQ_GEN application indicate that the bulk of classes declare three or less instance variables; the UGC_GK values are more evenly distributed. The differences would narrow if inherited variables were included in the analysis, since a higher percentage of SEQ_GEN classes inherit attributes (DIT > 0, see Figure 2). A high average coupled with little subclassing could indicate that abstractions are being broadly defined. A class with a high number of instance variables should be analyzed for possible reorganization into a subtree of related classes. Reusability is enhanced also.

Number of Methods: The number of instance methods indicates the scope of behaviors that an object may exhibit. Each instance method is declared as a member function. As with variables, the measurement tool excludes inherited methods. This metric corresponds to the Chidamber and Kemerer metric weighted methods per class (WMC), with a weight of unity assigned each method. The averages and distributions for SEQ_GEN and UGC_GK are given in Table 2 and Figure 4, respectively. Their averages are similar, and the UGC_GK distribution is slightly skewed toward the higher values. Thresholds follow the same trend as for number of variables. However, higher numbers of methods are acceptable, since in addition to methods which manipulate variables, the class may define an extractor method for each variable (public method which yields the value of the private variable). In general, classes having a large number of methods are less reusable. Such classes are usually very specific and thus not prone to specialization through subclasses or adaptation to a different application.

Method Weights: Each method is defined a weighted based on the number of non-comment source statements (NCSS) in its definition. The system wide averages are given in Table 3. The per-class sum of method weights is of greater interest, however, since the class is the building block of an object-oriented system. The averages and distributions of the sums are shown in Table 2 and Figure 5, respectively. The distribution curves are quite similar, and heavily skewed to the low end. Most classes contain a relatively small amount of source code, less than 100 NCSS, indicating effective decomposition of the application. The sum of method weights tends to correlate directly to the number of methods.

The following metrics are relevant to the measurement and analysis of method weights.

Non-Comment Source Statements: This is reflected in all metrics involving method weights. It was collected from C++ source code by counting the number of semi-colons (;), which serve as statement terminators, as well as curly braces ({ and }), which group statements in the fashion of begin-end.

Inline Methods: C++ provides method and function inlining to enhance runtime performance. Inlining is the compile-time substitution of a function body at the point of each call to it, which avoids the runtime overhead of context switching. A method is automatically inlined if defined within the class definition itself. Most inline methods are trivial, such as those which return the value of a private variable. The measurement tool counts each one as 3 NCSS. The use of inline functions will contribute to low method weight metrics, possibly to the point of hiding the complexity of the remaining methods. The ratio of inline to total methods is expressed as a percentage in Table 3.

Pure Virtual Methods: Such methods are literally defined to be null, but are used as place holders in the implementation of polymorphism in C++. They contribute nothing to the system size, but are counted as methods nonetheless. One expects to find a small number of such functions, concentrated in classes at or near the root of an inheritance tree. Their small numbers are reflected in the percentages given in Table 3.

Table 1. System Size Metrics.

METRIC	SEQ_GEN	UGC_GK	Ratio
Number of Classes	313	43	7.3 - 1
Number of Methods	2283	303	7.5 - 1
Sum of Method Weights	30345	7423	4.1 - 1

Table 2. Class Metric Averages.

METRIC	SEQ_GEN	UGC_GK
Number of Variables	2.7	6.1
Number of Methods	7.3	7.0
Method Weights	96.9	172.6

Table 3. Selected Method Metrics.

METRIC	SEQ_GEN	UGC_GK
Average NCSS	13.7	26.3
% Inline Methods	53.9	12.6
% Pure Virtual Methods	2.7	1.4

Figure 1. Metric Distribution: Number of Child Classes.

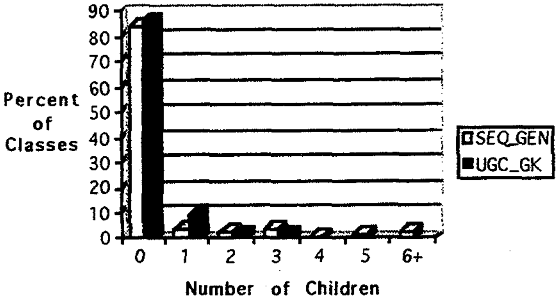


Figure 2. Metric Distribution: Depth In Tree.

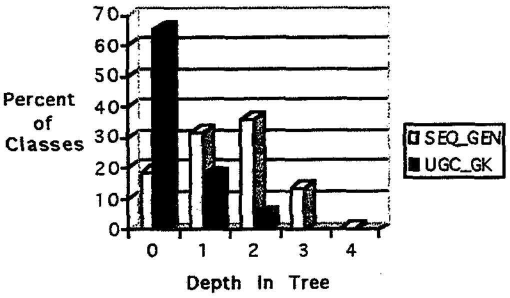


Figure 3. Metric Distribution: Number of Variables.

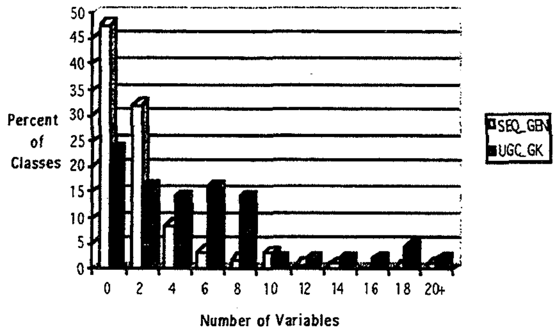


Figure 4. Metric Distribution: Number of Methods.

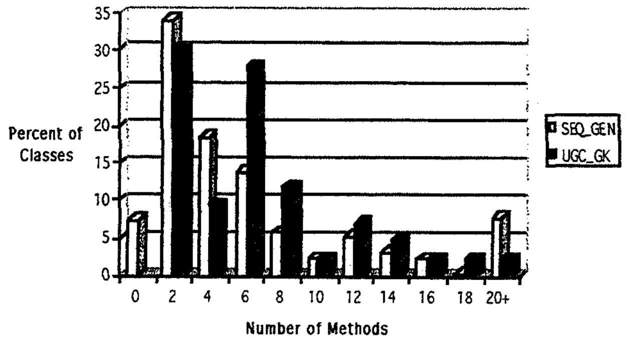
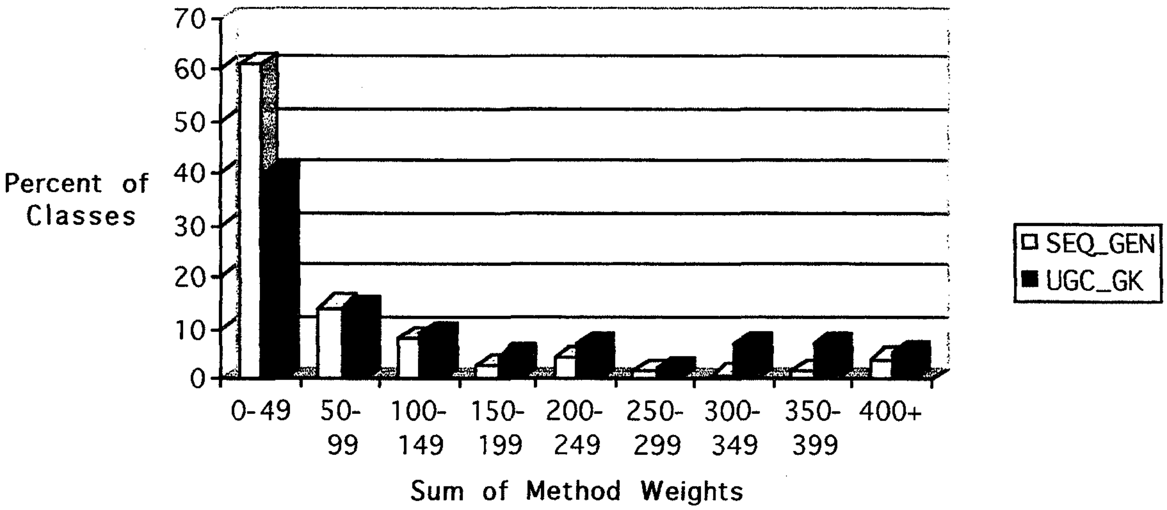


Figure 5. Metric Distribution: Method Weights.



6. CONCLUSIONS

Although our efforts in this research are still at an early stage, some preliminary conclusions can be drawn. First, we are encouraged by the results achieved so far, and intend to continue the endeavor. Second, metrics selected from the Chidamber and Kemerer suite may be applied and adapted to a variety of software quality assurance goals, in this case size estimation and reusability prediction. Third, care should be taken in the selection of the system size metric, if a single such metric is desired. Fourth, reusability prediction through metric collection and analysis is a difficult task.

Our metric analysis of the SEQ_GEN and UGC_GK systems shows results consistent with those of similar analyses reported in the literature. The relatively small size of UGC_GK accounts for most of the observed differences in metric values and distributions. More work needs to be done, and more systems studied, before reasonable conclusions about reusability can be made. We also conclude from our analysis that sound methods of object-oriented design were applied to both systems. This provides some evidence of reusability based on the inherent qualities of well-designed classes. Neither system is completely object-oriented; size metrics for free function (not associated with a class) were collected but not reported here.

The metrics and analysis described here may be applied to other object-oriented systems given detailed design documents or source code. Confidence in our methods will grow as more applications are analyzed, but even these preliminary results should be useful to those concerned with quality assurance as well as other aspects of the development of object-oriented software for technical applications.

ACKNOWLEDGMENTS

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The principal author was supported in part by a 1995 NASA/ASEE Summer Faculty Fellowship.

REFERENCES

Abreu, F.B., and R. Carapuca, Candidate metrics for object-oriented software within a taxonomy framework, *Journal of Systems and Software*, Vol. 24, 1994, 87-96.

Chidamber, S.R., and C.F. Kemerer, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994, 476-493.

Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.

Halstead, M.H., *Elements of Software Science*, Elsevier North-Holland, New York, 1977.

Li, W., and S. Henry, Object-oriented metrics that predict maintainability, *Journal of Systems and Software*, Vol. 23, 1993, 111-122.

Lorenz, M., and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.

McCabe, T.J., A complexity measure, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, 308-20, 1976.

Rosenberg, L.H., Metrics for risk assessment, software quality, and process improvement, *The Third Symposium on Foundations of Software Engineering*, October 1995.

Sanderson, D.P., "Overview of oo_metrics: A tool for measuring object-oriented metrics for C++," unpublished report funded by NASA/ASEE Summer Faculty Fellowship. Jet Propulsion Laboratory, August 1995.

Object-Oriented Software Metrics for Predicting Reusability and Estimating Size

Peter Sanderson
Southwest Missouri State University

T.-L. Tran, J. S. Sherif, S. S. Lee
Jet Propulsion Laboratory

Outline

- Introduction
- Background
- Description of Metrics
- Systems Measured
- Metric Results
- Conclusions

Introduction

- Metrics for Object-Oriented Systems
- Metric Project Goals:
  - Size Estimation
  - Reusability Prediction

Background

- Lexicon of object-oriented terms
  - Class
  - Instance Variable
  - Instance Method
  - Inheritance
- Chidamber/Kemerer metrics suite

System Size Estimation Metrics

- Number of Classes
- Total Number of Methods
- Sum of Method Weights
  - Non-Comment Source Statements

All are based on "sum of class sizes."

Class Reusability Metrics

- Number of Child Classes
- Depth in Inheritance Tree
- Coupling Between Classes (not measured)
- Number of Instance Variables
- Number of Instance Methods
- Method Weights

Systems Measured

- Sequence Generator (SEQ_GEN)
  - subsystem of Advanced Multimission Operations System
- Microwave Generic Controller (UGC_GK)
  - controls 34-meter Beam Waveguide Antenna
  - measured Generic Kernel software
- Both implemented in C++
- Measured using author's oo_metrics tool

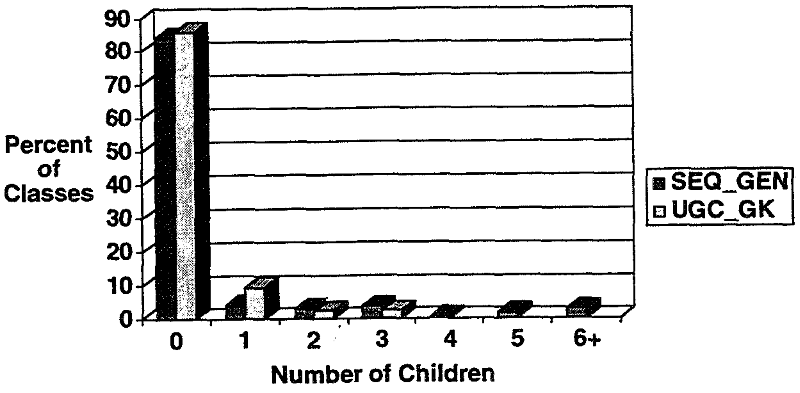
Results: Size Metrics

METRIC	SEQ_GEN	UGC_GK
Number of Classes	313	43
Total Number of Methods	2283	303
Sum of Method Weights	30345	7423
Class Size (average)		
Number of Methods	7.3	7.0
Method Weights	96.9	172.6
Method Weight (average)		
NCSS	13.7	26.3

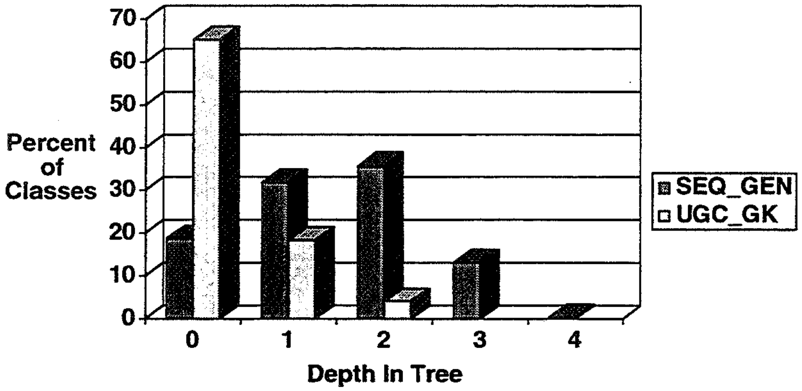
Results: Reusability Metrics

- Class metrics summarized on graphs
- Distribution characterizes system
- Extremes call attention to individual classes

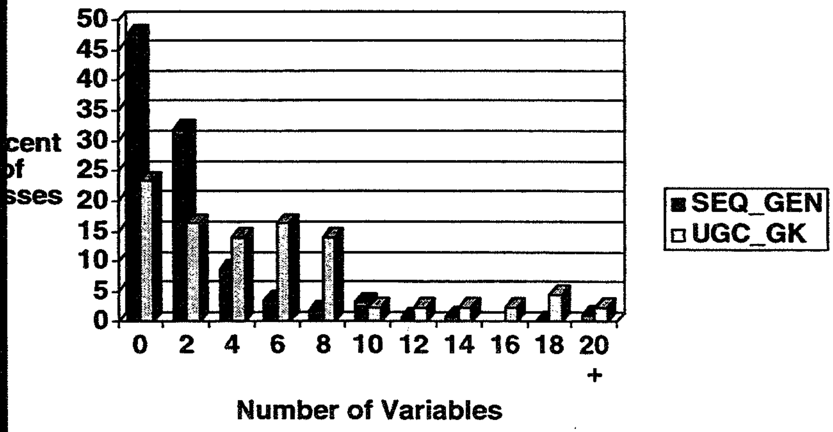
Number of Child Classes



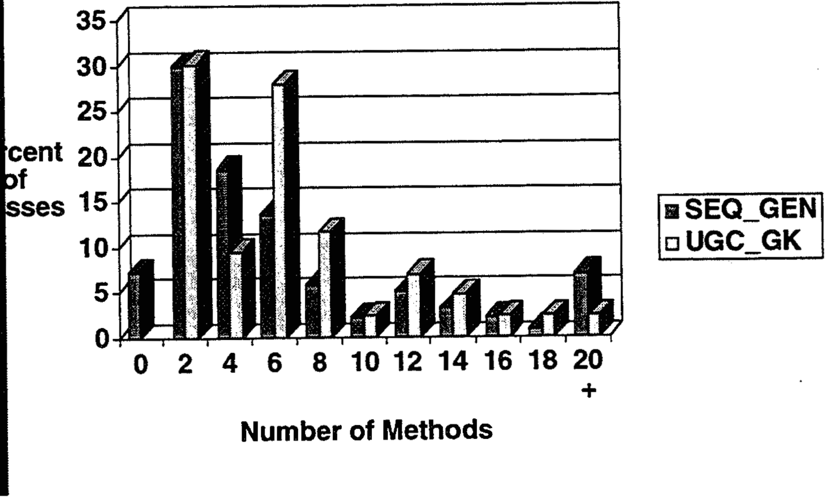
Depth in Inheritance Tree



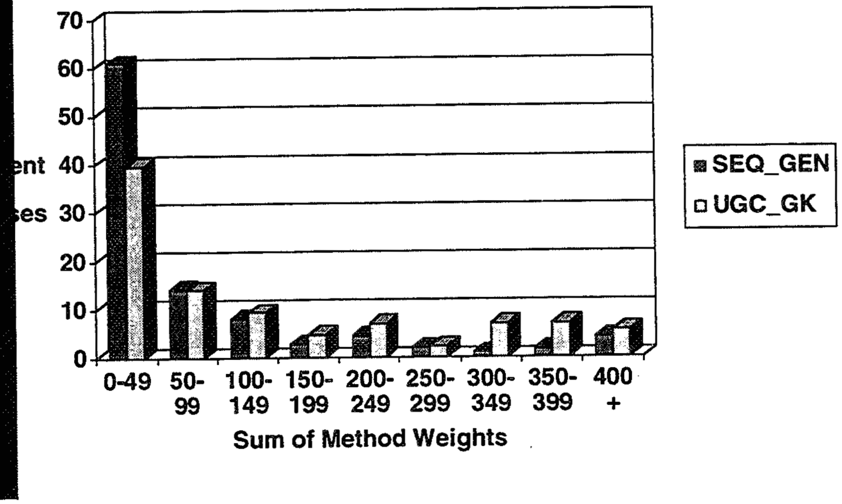
Number of Instance Variables



Number of Instance Methods



Method Weights per Class



Conclusions

- Preliminary results are encouraging.
- Size metric should be selected carefully.
- Reusability is difficult to predict using metrics.
- Systems studied are consistent with others reported in literature.

Improving the Software Testing Process in NASA's Software Engineering Laboratory

Sharon Waligora
Computer Sciences Corporation
10110 Aerospace Road
Lanham-Seabrook, MD 20706
(301) 794-1744
swaligor@csc.com

Richard Coon
Computer Sciences Corporation
10110 Aerospace Road
Lanham-Seabrook, MD 20706
(301) 794-1979
rcoon1@csc.com

In 1992, the Software Engineering Laboratory (SEL) introduced a major change to the system testing process used to develop mission ground support systems in NASA Goddard's Flight Dynamics Division. This process change replaced two sequential functional testing phases (system testing and acceptance testing) with a single functional testing phase performed by an independent test team; functional system testing by the software developers was eliminated. To date, nine projects have been completed using the new independent testing process. The SEL recently conducted a study to determine the value of the new testing process, by assessing the impact of the testing process change on the delivered products and overall project performance. This paper reports the results of this study; it presents quantitative evidence that this streamlined independent testing approach has improved testing efficiency.

This paper presents the results of a study that the authors conducted in 1995 to assess the value of a major system testing process change that was introduced in 1992 in the Flight Dynamics Division at NASA Goddard. It first presents background information to provide the context for the study. This includes information about the SEL and its project environment and highlights some key improvements that had taken place before the testing change was introduced. It then describes the testing process change and our goals and expectations for the new process. The bulk of the paper presents quantitative results that clearly show the impact of the new testing process on recent projects. These results are followed by the conclusions drawn from this study.

Background

The Software Engineering Laboratory (SEL) is a partnership of NASA Goddard's Flight Dynamics Division, its major software contractor, Computer Sciences Corporation, and the University of Maryland's Department of Computer Science. The SEL is responsible for the management and continuous improvement of the software engineering processes used on Flight Dynamics Division projects.

The SEL process improvement approach [1] is based on the Quality Improvement Paradigm [2], in which process changes and new technologies are 1) selected based on a solid understanding of organization characteristics, needs, and 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 facilitated and measured significant improvement in project performance and product quality [1].

The Flight Dynamics Division primarily builds software systems that provide ground-based flight dynamics support for scientific satellites. The projects included in this study cover the period from 1985 through 1995 and fall into two sets: ground systems and simulators. Ground systems are midsize systems that average around 250 thousand lines of code (KSLOC). Ground system projects typically last approximately 2 years. Most of the systems have been built in FORTRAN on mainframes, but recent projects contain some subsystems written in C and C++ on workstations. The simulators are smaller systems averaging around 60 KSLOC. These are much smaller projects that last between 1 and 1.5 years. Most of them have been built in Ada on a VAX computer. Simulators provide the test data for the ground systems. The project characteristics of these systems are shown in Table 1.

Table 1. Characteristics of Flight Dynamics Division 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

Before delving into the testing process change and its results, it is important to understand where the SEL was in its improvement process when the testing improvement initiative began. Previously, the SEL had largely focused its efforts on activities of the design and implementation phases of the life cycle. Experimentation with object-oriented analysis and design had led to a threefold increase in software reuse [3]. Projects were regularly achieving 60% to 90% reuse in the early 1990s. This, in turn, had significantly reduced the cost to deliver systems by reducing the amount of code that needed to be developed. However, it is interesting to note that the cost to develop new code had remained relatively constant. The SEL had also done extensive studies of unit testing techniques and code inspections that led to reduced development error rates (75% reduction).

By 1992, system testing had become the largest part of the software development job, as shown in Figure 1. With higher reuse, less project effort was required in design and coding. In fact, more time was now being spent doing testing than doing design and coding combined. From the business point of view, testing was a natural target for the next major improvement initiative. Also, the testers had pointed out that although the system testing process was effective, in that it produced high-quality systems, they felt it was somewhat inefficient. The SEL had also learned quite a bit about the value of independent testing from experimentation with the Cleanroom methodology. Thus, the SEL turned its attention to improving the system testing process.

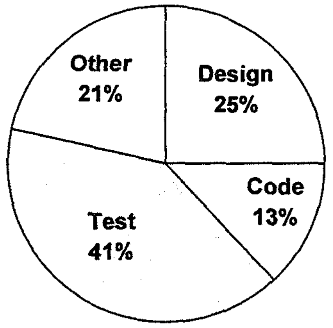


Figure 1. Project Effort By Activity

System Testing Process Changes

So, in 1992, the SEL began its initiative to improve the system testing process. The primary goals of this initiative were to reduce the cost of testing and to shorten the project cycle time without degrading the quality of our delivered systems. Software developers, managers, and acceptance testers worked together to identify inefficiencies and weaknesses in the standard SEL testing approach and to propose improvements.

This group proposed a series of process changes and corresponding organizational changes to support them. The changes focused on eliminating redundancy in functional testing and identifying operational deficiencies (deficient requirements or software) earlier in the life cycle. The following subsections provide a high level description and comparison of the standard SEL testing approach (old testing process) and the new combined independent testing approach (new testing process).

Standard SEL Testing Process

Figure 2 presents the standard life cycle that had been used in the SEL for many years [4]. It is a typical waterfall life cycle in which the system is fully implemented before any system testing begins. There are two sequential functional testing phases. In the system testing phase, the developers test the system against their interpretation of the written requirements. When they are satisfied, they hand it over to a separate group of testers representing the users, who perform another round of functional testing on the system, called acceptance testing. These testers have flight dynamics application expertise and operational experience. They test the system to be sure that it meets operational requirements (i.e., that it can successfully support the mission). Thus, two very similar sets of functional tests are run, with the second set being most realistic for the mission. When the system passes acceptance testing, it is delivered to the users.

Design	Code Build 1	Code Build 2	Code Build 3	System Testing	Acceptance Testing

Figure 2. Standard SEL Testing Approach

This approach has several disadvantages. First, operational deficiencies are not uncovered until very late in the development life cycle. Second, the separate, sequential functional testing phases are time consuming and somewhat redundant. Thus, the value of separate functional testing phases is questionable.

New Combined Independent Testing Approach

The new testing approach, shown in Figure 3, involved both an organizational change and a process change. First, we combined system testing and acceptance testing into a single independent testing phase. Second, we created an independent test group within the software engineering organization to do all functional testing and staffed it with people who have flight dynamics expertise and operational experience. Third, we began independent testing earlier, as soon as the first build is completed. The testers test the most recently completed build while the developers move on and implement the next build. The developers are responsible for integration testing of each build before it is delivered to the independent testers. When the system is complete, the testers perform end-to-end testing of the full system for a short period.

Developers	Design	Code Build 1	Code Build 2	. . .	Code Build N	Cleanup
Testers	Prepare Test Plans		Test Build 1	Test Builds 1 & 2	. . .	Test Full System
			|←	Acceptance Testing		→|

Figure 3. New Combined Independent Testing Approach

Comparison of Testing Approaches

Figure 4 highlights the key differences between the two approaches. The new approach is shown on the top and the standard or old testing approach is shown on the bottom. Key differences are as follows:

- In the new approach, system testing begins much earlier, about halfway through the development life cycle. This allows more calendar time for testing the integrated system and enables the testers to identify operational deficiencies earlier in the project.
- In the new approach, the system is completed very late in the life cycle. This allows the developers more time to resolve incomplete requirements and to respond to the changing requirements that are inevitable in our business.

- In both cases, the software is placed under configuration control near the completion of build 1. However, in the new approach, system testing begins right after the start of configuration control. Therefore, errors found in the configured code would be reported during the testing phase rather than distributed over the code and testing phases as they were with standard approach.

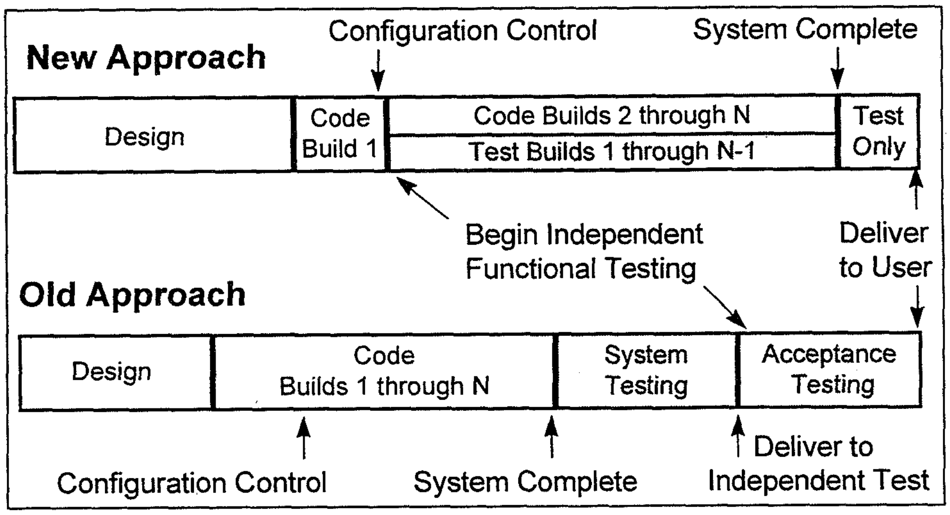


Figure 4. Comparison of Testing Approaches

Quantitative Analysis

To assess the overall impact, or value, of the new testing approach, we did a quantitative analysis of key process and product measures from projects before and after the introduction of the new testing process. We computed new baseline measures[1] for projects that used the new testing approach and compared them with baseline measures of comparable earlier projects.

Sova used a different approach in a similar study [5] in which he compared three testing approaches used in the Flight Dynamics Division, namely the standard SEL approach, the modified or combined independent testing approach, and the Cleanroom statistical testing approach. His selection criteria carefully screened the projects to be sure that the project samples were highly comparable except for the testing process used. His study focused on one subsystem common to all ground systems that was minimally affected by reuse. Six projects were included in his study; two projects for each testing approach. In contrast, our analysis included ground systems and simulator projects completed during each baseline period, excluding only those projects where special circumstances caused them to be unrepresentative samples.

To date, five ground systems and four simulators have been completed using the new independent testing process. The study compared software engineering measures for this recent project set (projects completed between 1993 and 1995) with those from two earlier baseline time periods. The first includes projects between 1985 and 1989, when a fairly traditional

[1] A baseline measurement is the average of the projects (measurements) in a particular baseline time period; it represents the typical project measurement from that time period.

waterfall process was used and projects averaged 20% reuse. The second period covers 1990 through 1992, when projects were regularly achieving between 60% and 90% reuse and the process had been tailored to accommodate high levels of verbatim reuse. All projects in both of the earlier baseline periods used the standard SEL (system and acceptance) testing process described above. We separated the earlier projects into two baseline sets according to reuse levels so that we could more clearly see the effect of the testing process change from one high reuse period to another. Table 2 shows the characteristics and number of projects included in each baseline period.

Table 2. Characteristics of Baseline Periods

Time Period	Testing Approach	Reuse Level	Systems Included
1985 - 1989	Standard Testing (ST)	Low Reuse (LR)	4 Ground Systems 5 Simulators
1990 - 1993	Standard Testing (ST)	High Reuse (HR)	3 Ground Systems 3 Simulators
1993 - 1995	Independent Testing (IT)	High Reuse (HR)	5 Ground Systems 4 Simulators

Process Change Confirmed

Software development activity data, shown in Figure 5, clearly confirms that a process change has taken place. Displayed here are two donut charts. In each case, the inside donut represents the standard testing approach and the outside one represents the new testing approach. The left chart shows the relative percent of effort performed by the developer organization vs. the independent testing organization. The chart on the right shows the percentage of project effort spent doing development activities (design and coding) vs. testing activities regardless of organization. In both charts the testing portion is shaded.

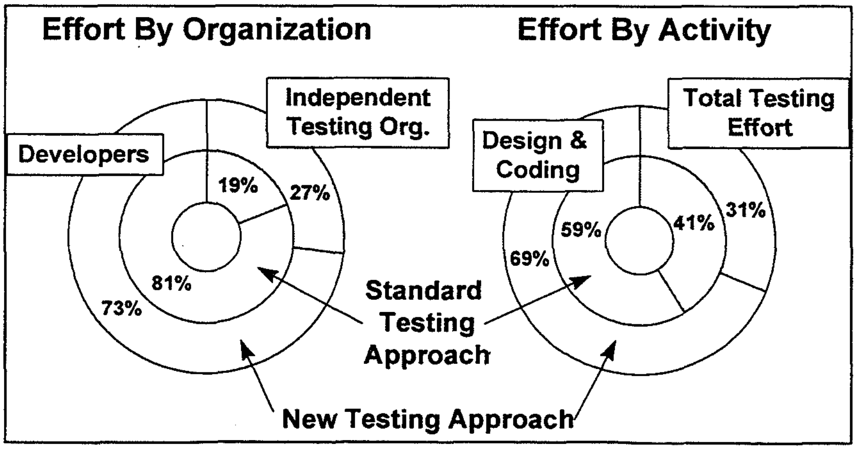


Figure 5. Distribution of Project Effort

Overall system testing effort declined from 41% to 31% of the total project effort (a 25% reduction). Although recent projects spent a smaller portion of the total effort doing testing overall than earlier projects did, the independent testing organization now contributes a larger part of the total effort (an increase from 19% to 27%). The difference between the total testing effort and the amount spent by the independent testers represents the amount of testing done by the software developers. This data indicates that currently software developers are spending only 4% of the total effort on integration testing as compared with 22% on system testing activities(including integration testing) for the earlier systems. This confirms a major shift in the process used and the responsibility for testing systems.

Cost To Deliver a Line of Code

Figure 6 shows the average cost to deliver a line of code[2] for each of the three baseline periods. The labels under the bars indicate the testing approach and the level of reuse during each period (see Table 2). Each bar is divided into three parts showing the portion of the cost attributable to the various software engineering activities. The bottom portion represents design and coding, the middle indicates the amount of system or integration testing done by the developers, and the top section indicates the portion of the effort spent doing independent or acceptance testing.

As you can see, there was a significant reduction in the overall cost to deliver a line of code with the increase in reuse. The introduction of the new testing approach in the third baseline period had a smaller impact (15% reduction) on the cost to deliver.

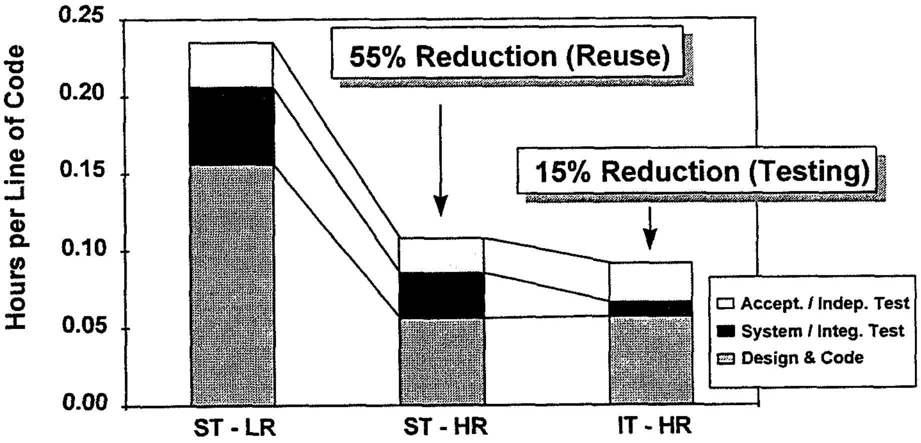


Figure 6. Cost to Deliver A Line of Code

Cost To Develop a New Line of Code

By removing the effects of reuse, we see a different picture. Figure 7 is similar to Figure 6 except that it shows only the cost to develop new code[3]. As you can see, the cost to develop a new line of code increased somewhat in the middle time period, along with a fairly large increase in the cost of testing a new line of code.

[2] Cost to Deliver = Total Project Effort / Total Lines of Code

[3] Cost to Develop a New Line of Code = Total Project Effort / New and Modified Lines of Code

After the introduction of independent testing, effort measures show a 23% reduction in actual cost to develop a new line of code. A closer look shows that most of the savings have occurred in testing activities, where a reduction in developer test effort (78%) and a 33% rise in independent testing effort (33%) together net a 40% reduction in overall testing cost per new line of code. This is significant because the SEL had previously never seen a decrease in the cost to develop new code despite large reductions in the cost to deliver systems as a result of high reuse.

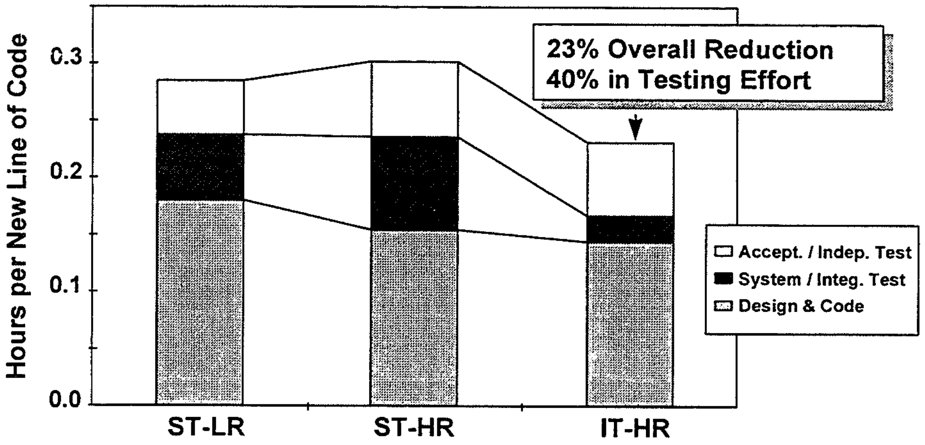


Figure 7. Cost To Develop A New Line of Code

Separating the data shown in Figure 7 by project type uncovered a difference in cost savings. As shown in Figure 8, ground system development projects reaped a significant 35% savings, almost entirely due to a reduction in testing cost, while simulators experienced only a 10% overall reduction in cost with minimal savings in testing. Close examination of project histories revealed that the simulators tended to have fewer builds and a smaller overlap between coding and testing. Also, simulators are now tested under more schedule pressure because they are needed earlier to begin testing the ground systems. This data seems to indicate that cost may be negatively affected by schedule pressure. Further analysis is needed to clarify this.

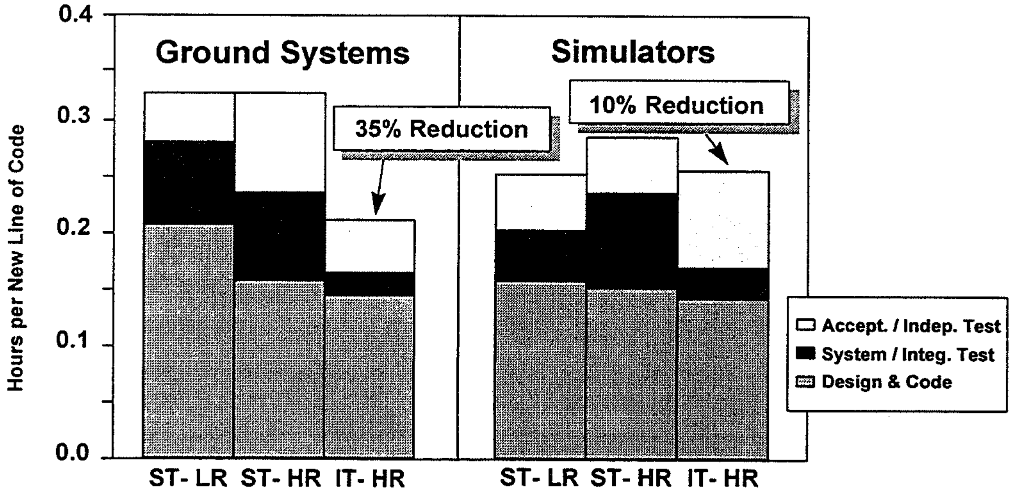


Figure 8. Cost To Develop A New Line of Code by Project Type

Shorter Cycle Time to Delivery

Schedule data for the recent projects, presented in Figure 9, show a slight reduction in average project duration for ground systems and a 29% reduction for simulators, when compared to the middle (high reuse) time period. Notice that the schedule impact on each type of project is the opposite of the impact on cost (shown in Figure 8). Simulators saved time, but not cost; while ground systems saved cost, but not time. It appears that the ground systems projects have used the productivity gain from the new testing process to reduce staff, while the simulator projects have used it to reduce schedule. Further analysis is required to fully understand the trade-off between cost and schedule and its relationship to the testing process, but is clear that the new testing process has helped projects meet their various objectives.

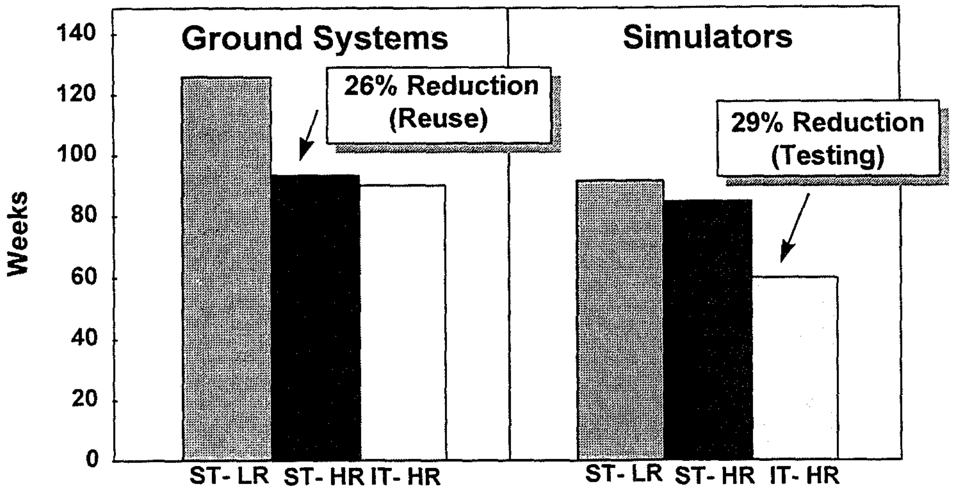


Figure 9. Average Project Duration by Project Type

System Quality

Development error rates of the recent systems, shown in Figure 10, also decreased when compared with the earlier systems. The average error rate on the recent projects is 1.5 errors per KSLOC. This is down from 4.3 errors per KSLOC and 2.8 errors per KSLOC for the two earlier baseline periods. The distribution of errors by phase (indicated by the shaded subsections of each bar in Figure 10) reveals that very few errors are now reported by the software developers (during the implementation phase). Although more errors are uncovered by the independent test team using the new approach than when doing standard acceptance testing, fewer errors were reported altogether during the system testing portions of the life cycle. However, it is

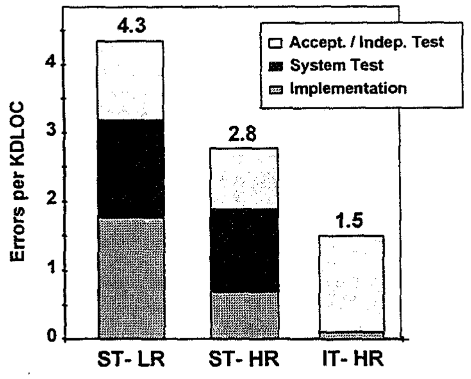


Figure 10. Development Error Rates

unclear from this data whether the product quality is improving or whether the independent testers are not finding all of the errors. A better indicator of system quality would be the error rate during the first year of operational use. Unfortunately, records of operational errors were not kept for systems in the early baseline period and the recent systems, tested using the new testing method, are now just beginning to be used operationally. Thus, it is too early to judge whether the new testing process maintains quality.

Conclusions

Based on our quantitative analysis, we conclude that the combined independent testing approach is beneficial and should be adopted as the standard testing approach to be used on future projects in the Flight Dynamics Division. Our analysis revealed that project performance improved without sacrificing quality on projects using the new testing approach. Specific performance factors are addressed below.

- *Cost*: We conclude that the new testing approach reduces project cost. There was significantly reduced testing effort on projects using the new approach, which consistently contributed to overall project cost savings.
- *Cycle Time:* Because cycle time improvement varied by project type, we conclude that shorter cycle times are possible, but not guaranteed when using the new approach. There appears to be a tradeoff between cycle time and cost; i.e., the shorter the test phase, the more it will cost.
- *Quality:* It is too early to determine the impact of the new testing approach on product quality. Although project data continue to show a decline in the development error rates, operational error rates are not yet available for most of the recent systems.

One additional benefit of the new testing approach is the longer overlapped periods of implementation and testing. This allows more calendar time for both development and testing. This stretched-out development time period provides the flexibility needed to deal efficiently with late-coming requirements, thereby reducing rework. The stretched-out testing time period allows for a smaller team of testers to test each system, thereby reducing the learning curve and capitalizing on growing mission knowledge as testing proceeds.

The results of this study are similar to those found by Sova [5]. His study found the independent testing approach (referred to as the modified approach) to be most efficient and produce the lowest fault density. Thus, both the broad brush analysis of baseline comparisons and the detailed comparison of carefully selected samples confirm that the new independent system testing approach is beneficial and should be adopted as the Flight Dynamics Division standard testing process.

References

1. McGarry, F., G. Page, V. Basili, et al., *An Overview of the Software Engineering Laboratory*, Software Engineering Laboratory, SEL-94-005, December 1994
2. Basili, V., "Quantitative Evaluation of a Software Engineering Methodology," *Proceedings of the First Pan Pacific Computer Conference*, Melborne, Australia, September 1985

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, March 1995

4. Landis, L., S. Waligora, F. McGarry, et al., *Recommended Approach to Software Development (Revision 3)*, Software Engineering Laboratory, June 1992

5. Sova Jr., D., "A Study of Software Testing Methodologies Within the Software Engineering Laboratory", M.S. Thesis, University of Maryland, 1995

Improving the Software Testing Process in the Software Engineering Laboratory

Sharon Waligora and Richard Coon
Computer Sciences Corporation

Topics

- Background
- Process Changes
- Quantitative Results
- Conclusions

SEL Environment

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

SEW95 3

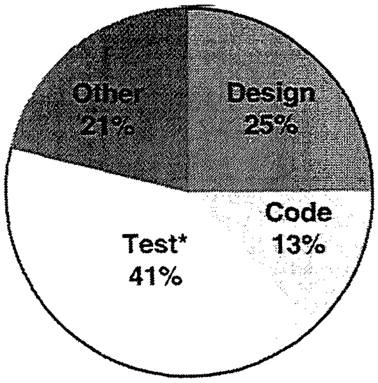
Previous SEL Improvements

- Focused on design and implementation activities
  - Object-oriented techniques
  - Code inspections and unit testing
- Increased reuse of code from 20% to ~80%
- Reduced cost to deliver systems by 55%
  - Cost to develop a new line of code remained the same
- Reduced error rates by 75%

SEW95 4

Status of Testing in 1992

Effort by Activity

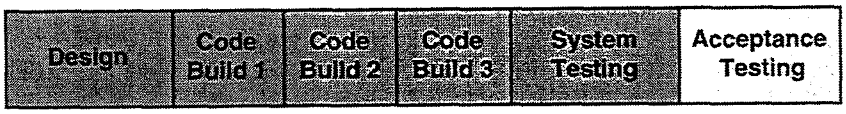


* Test includes integration testing, system testing, and acceptance testing

- Testing had become the largest portion of project effort
- Developers and testers felt that the standard testing process was effective, but inefficient
- Cleanroom experiments showed the value of using independent testers

SEW95 5

Standard SEL Testing Approach


Design | Code Build 1 | Code Build 2 | Code Build 3 | System Testing | Acceptance Testing

- System completely implemented before system testing begins
- Two sequential functional testing phases
- Acceptance testing done by separate organization representing the users
  - Testers have operational flight dynamics expertise

SEW95 6

Improvement Goals and Expectations

- Improvement goals
  - Eliminate redundant functional testing
  - Identify operational deficiencies earlier in life cycle
- Expected results
  - Reduce cost of testing
  - Shorten cycle time
  - Maintain quality

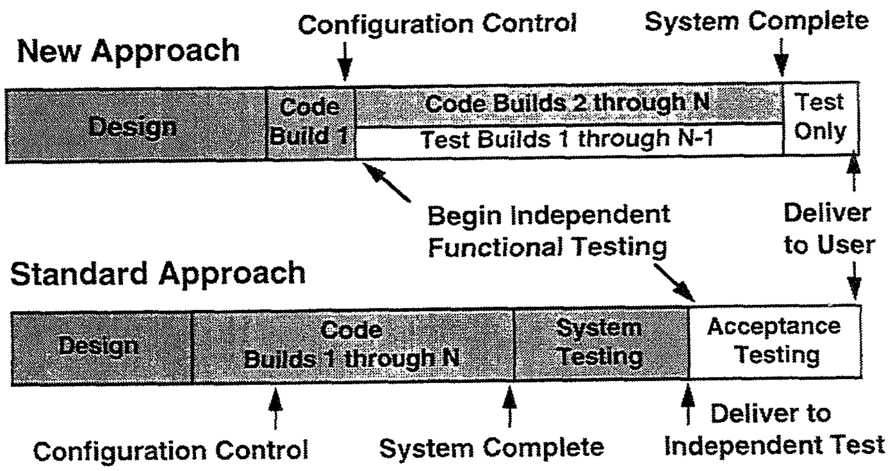
New Combined Independent Testing Approach

Developers	Design	Code Build 1	Code Build 2	. . .	Code Build N	Cleanup
Testers	Prepare Test Plans		Test Build 1	Test Builds 1&2	. . .	Test Full System

[redacted]

- Independent testing group within the software engineering organization performs all functional testing
- Developers perform integration testing only
- Incremental builds are delivered for testing

Comparison of Testing Approaches



New Approach

Configuration Control

System Complete

Design

Code Build 1

Code Builds 2 through N

Test Builds 1 through N-1

Test Only

Begin Independent Functional Testing

Deliver to User

Standard Approach

Design

Code Builds 1 through N

System Testing

Acceptance Testing

Configuration Control

System Complete

Deliver to Independent Test

SEW95 9

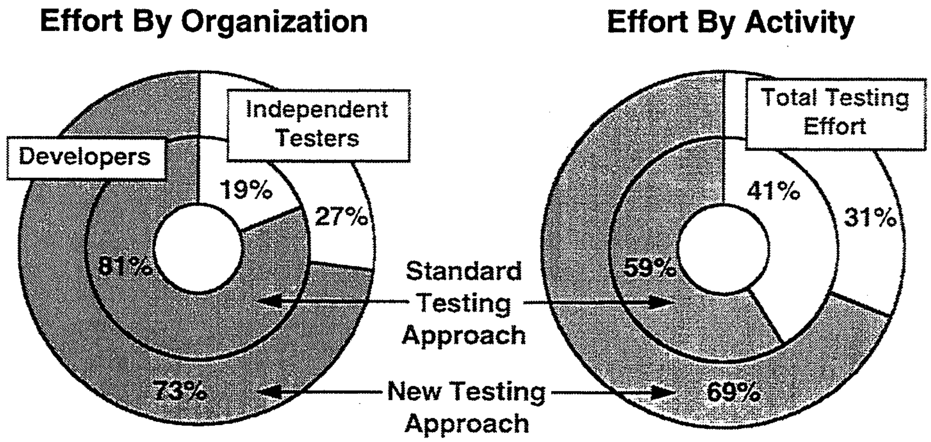
Quantitative Analysis

- Comparison of projects before and after introduction of new testing approach in late 1992
- Three baseline time periods

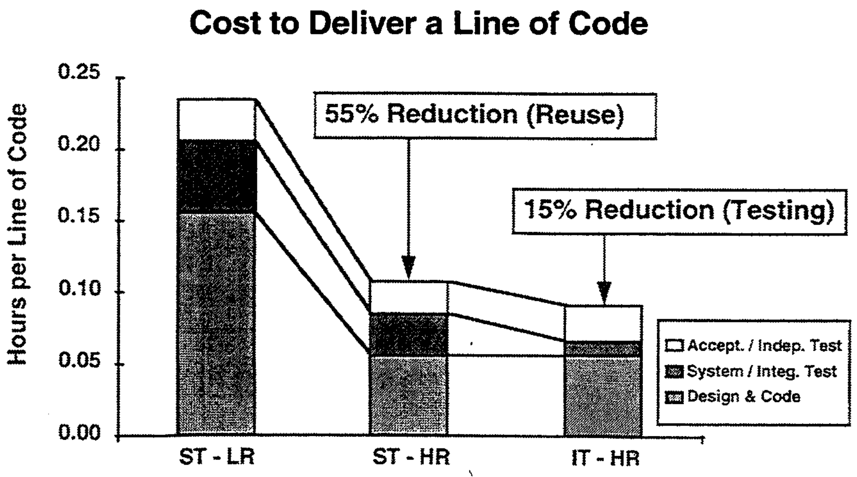
1985-1989	Standard Testing (ST)	Low Reuse (LR)	4 Ground Systems 5 Simulators
1990-1993	Standard Testing (ST)	High Reuse (HR)	3 Ground Systems 3 Simulators
1993-1995	Independent Testing (IT)	High Reuse (HR)	5 Ground Systems 4 Simulators

SEW95 10

Testing Effort Shifted to Independent Testers

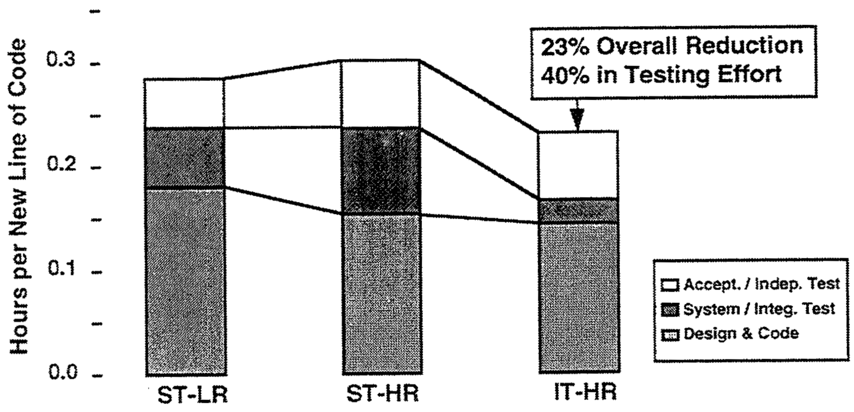


Cost To Deliver a Line of Code Decreased Slightly



Cost To Develop a New Line of Code Decreased Significantly

Cost to Develop a New Line of Code

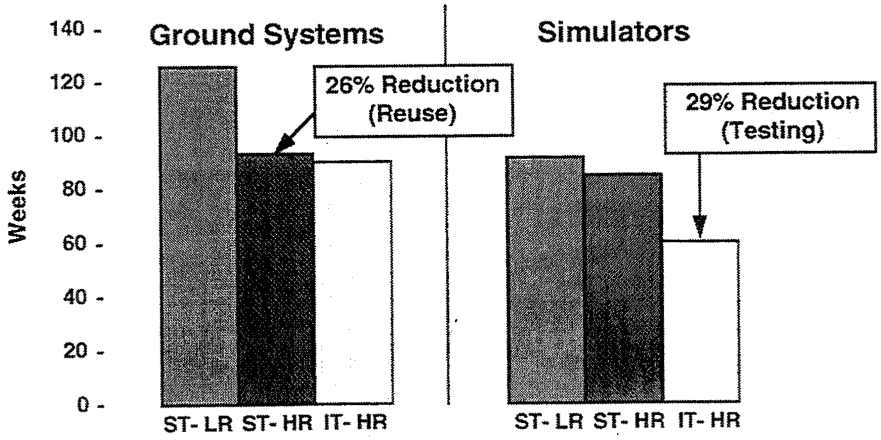


23% Overall Reduction
40% in Testing Effort

SEW95 13

Cycle Time Improved on Some Projects

Actual Change in Project Duration



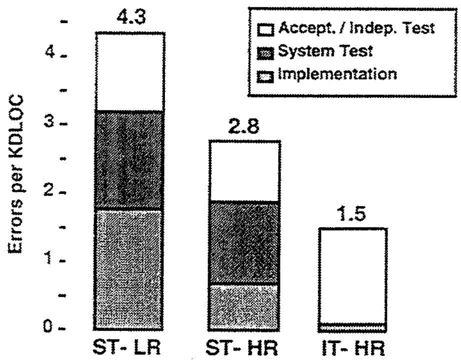
26% Reduction (Reuse)

29% Reduction (Testing)

SEW95 14

Quality Has Been Maintained

Error Rates By Phase



- Development error rates have continued to decrease
- Most errors are found in independent testing

Conclusions

- The new combined independent testing approach significantly reduces testing effort
- Shorter cycle times are possible, but there appears to be a tradeoff with cost
- Stretched out development time period provides flexibility needed to efficiently deal with late-coming requirements
- No degradation of quality is evident

S16-61
45012
CLOSE

How Do Formal Methods Affect Code Quality?

Shari Lawrence Pfleeger
Systems/Software, Inc.
4519 Davenport St. NW
Washington, DC 20016-4415
USA
phone: 202 244-3740

email: slpfleeger@aol.com

Les Hatton
Programming Research Ltd.
Glenbrook House, 1/11 Molesey Road
Hersham, Surrey KT12 4RH
England
+44 1932 888080
+44 1932 888081 (fax)
les_hatton@prqa.co.uk

16P.

360707

Formal methods are advocated on many projects, in the hope that their use will improve the quality of the resulting software. However, to date there has been little quantitative evidence of their effectiveness, especially for safety-critical applications. We examined the code and development records for a large air traffic control support system to see if the use of formal methods made a measurable difference. In this case study, we show that formal specification, in concert with thorough unit testing and careful reviews, can lead to high-quality code. However, formal specification on its own may not have achieved this goal; the data show a clear, direct relationship between the number of developers and the number of faults.

As Anthony Hall describes in a paper soon to appear in *IEEE Software*, Praxis built an air traffic control information system for the UK Civil Aviation Authority in the early 1990s using a variety of formal methods. The Central Control Function Display Information System (CDIS) provides controllers at the London Air Traffic Control Centre with key information, allowing them to manipulate the approach sequence of aircraft for the one of the busiest airspaces in the world. Hall describes the system function and architecture, making it clear that formal methods were an appealing technique for ensuring the quality of the CDIS software.

Praxis had used formal methods before, but not to the extent used in CDIS. Several different formal methods were involved in the development. For reasons of clarity described in detail by Hall, Praxis decided to do a complete, top-level formal specification of critical system elements using VDM. The development team found that this use of formal notation to capture essential CDIS operations improved their understanding of the requirements.

The abstract specification was written in a formal language, similar to VDM. The user interface definitions, derived from a prototyping exercise, were expressed as pictures, text and state-transition diagrams. The concurrency specification was a mixture of data flow diagrams and formal notation using Robin Milner's calculus of communicating sequential processes (CCS) technique. The abstract specification was by far the largest document, and all other documents were linked to its definitions.

The overall CDIS design was described in a *design overview*, containing the overall system architecture and design rationale. Then, each of the four major parts of the system had its own design document.

1. The application code was designed by writing VDM specifications of the application modules, created as refinements of the core specification.
2. The user interface code was designed informally, using pseudocode for each window class.
3. The processes needed to achieve concurrency and invoke the application code were defined as finite state machines.
4. The local area network software was designed formally, using a mixture of VDM and CCS. Because this area was particularly difficult, some formal proofs were done to find faults in the design.

As Hall points out, "While the three kinds of specification were three different *views* of the same thing, the four different designs (excluding the overview) were designs for different *parts* of the software." Thus, formal methods were involved in three places in the design: VDM for the application modules, finite state machines for the processes, and VDM with CCS for the LAN. For this reason, we can categorize any code in CDIS as being influenced by one of four design types: VDM, FSM, VDM/CCS or informally-designed.

Of the almost 200,000 lines of code delivered to the Civil Aviation Authority, the VDM/CCS-derived code (that is, the local area network software) was designed and implemented by a team of two developers. Most of the FSM work (including the links to external systems) was designed by one person. The graphical user interface code, which comprised most of the informally-developed programs, was developed by a team of four people. The VDM-only designs have as many as ten different authors, though several took responsibility for small areas of the overall system; the number of people who wrote the code from these designs was greater.

At the same time that CDIS was being developed, the British Department of Trade and Industry and the Science and Engineering Research Council funded a project to investigate the effectiveness of software engineering standards. Called SMARTIE (Standards and Methods Assessment using Rigorous Techniques in Industrial Environments) and led by the Centre for Software Reliability at City University, the collaborative academic-industrial partnership defined a framework for assessing standards and performed several case studies to investigate the effectiveness of particular standards in actual development environments. Shari Lawrence Pfleeger, Norman Fenton and Stella Page reported on the initial results of SMARTIE in the September 1994 issue of *IEEE Computer*.

Praxis offered the SMARTIE researchers the opportunity to examine CDIS development data to determine if the use of various formal methods had a positive effect on the resulting code. The CDIS development team had kept careful records of all faults reported during in-house system testing, as well as after fielding the system. Comments from the Civil Aviation Authority and users of the system were very positive, and the next step was to determine whether the perceptions of the users were supported quantitatively.

The SMARTIE team had three basic questions to answer:

1. Did the formal methods make a quantitative difference to the code quality?
2. Was one formal method superior to another?
3. How could data collection and analysis be improved to make quality questions easier to answer?

To begin our investigation, we captured information about each of the over-three thousand fault reports that were generated from the end of 1990 to the middle of 1992, when the software was delivered to the Civil Aviation Authority. Next, we classified each module and document by the type of design that influenced it: VDM, VDM/CCS, FSM or informal methods. Then, we generated some summary numbers to get a general idea of how design type affected the number of fault reports that were issued.

If quality is measured by the number of changes needed to correct modules, then our results show no clear indication that formal design methods produced higher-quality code than informal ones. (However, when viewed in terms of the number of developers involved in producing each type of code, there is a clear relationship. For each of the VDM/CCS, FSM and VDM components, the fewer the developers, the fewer the faults.) We analyzed the documents in the same way, doing a causal analysis to determine which changes occurred because of specification problems, design problems and code problems. None of our analysis provides compelling evidence that formal design methods are better than informal, in terms of the number of faults located in each design type.

The analysis of fault records was supplemented by a static analysis of the delivered code. Since the CDIS code is written almost entirely in C, the code was audited by Programming Research Ltd. using QAC, an automated inspection toolset for the C language. In essence, this toolset detects reliance on unsafe features of these languages as documented in *Safer C* (McGraw-Hill, 1995), Les Hatton's guidelines for developing safety-critical systems.

Programming Research has audited millions of lines of code in C packages from around the world in the last few years; this code, representing a wide variety of application domains including many safety-critical systems, formed the population against which the static code analytical results were compared. The audit involved two key steps: analyzing each module for potential faults remaining, and calculating several structure and dependence measures to compare the modules with the larger population in the Programming Research database. The first step helps Praxis to understand what types of coding errors are missed in their testing process; this step is accompanied by a risk evaluation to assess the likelihood that each latent fault will cause a significant error. The second step compares the overall system with other systems written in C, to give a general indication of where Praxis code quality falls in the larger universe of developed systems.

We found that CDIS contains an unusually low proportion of components with high complexity compared to the population at large. In fact, the CDIS code is one of the simplest large packages yet encountered in terms of component complexity. The audit presents a picture of modules that have a very simple design and are very loosely coupled with one another. Since the several packages comprising CDIS exhibit the same characteristics, and since the design techniques were different for each package, the simplicity cannot be attributed to a particular design method, formal or informal. Instead, the simplicity seems likely to be a direct legacy of the specification rather than the design. Whether the simplicity results from the use of a formal method for specifying the system or from a more thorough than usual analysis of the specification (that is, an indirect result of the formal method) is not clear and requires further investigation. However, the simple modules with few inter-module dependencies suggest that unit testing of such components would be highly effective.

Consequently, we investigated unit testing techniques and results. Praxis provided us with data showing the types of faults discovered during development but before system testing. Of all the pre-delivery faults reported, 340 occurred during code review, 725 in unit testing and 2200 during system and acceptance testing (that is, the non-zero fault reports we analyzed above). The faults discovered during unit testing were found in informally-designed modules more often than in formally-designed ones; this relationship persists even when the number of faults is normalized by dividing by the number of modules in a given design type, suggesting that formal design may have helped to minimize errors or to aid discovery early in development. The thoroughness of pre-delivery testing is dramatically borne out by the differences in failures reported before and after release; only 273 problems were reported between delivery in 1992 and the end of our dataset in June 1994 (of which 147 were actual code faults), so the delivered code is approximately ten times less fault-prone after system testing.

In our experience, this distribution of faults across review, unit testing and system testing is unusual. Statistics reported in the literature suggest that code review is far more effective than unit testing, so more faults should be found in review than in unit testing. There may be many reasons for this aberration, including the possibility that the use of formal methods makes problems more visible during unit testing; we do not have the data to enable us to make this determination.

A look at the post-delivery failures and their relationship to formal methods is also instructive, showing that far fewer changes were required to formally-designed parts of the delivered system than to informally-designed parts. A similar picture of high quality can be seen when the non-zero post-delivery problems are viewed in terms of their severity categories and root causes. Only six problems were rated Category 1, and only one specification and one design problem have arisen since delivery; the remaining problems were minor.

The difference between pre-delivery and post-delivery faults implies a strong demarcation between classes of faults founds by the different methodologies used for reviewing and testing the system. In particular, the difference illustrates how formal methods have two effects, one direct and one indirect. The direct effect is the large reduction in departures of the working system from the requirements, as shown by the post-delivery failure spread. The indirect effect is the highly-testable system that resulted, allowing satisfaction of the requirement for 100% statement coverage. What is intriguing is that the informally-designed code was just as testable, as shown by the metrics generated by the static code analysis. It is not possible to tell from the data whether this effect was cultural (that is, was a result of the general Praxis emphasis on quality and repeatability) or the result of some other technology or attitude. The informally-designed code was tested in a different way from the formally-designed code, so the testing technique may have influenced the results in ways that are not captured in the data.

Given the results described above, we can draw several important conclusions from these quantitative analyses performed on the Praxis code.

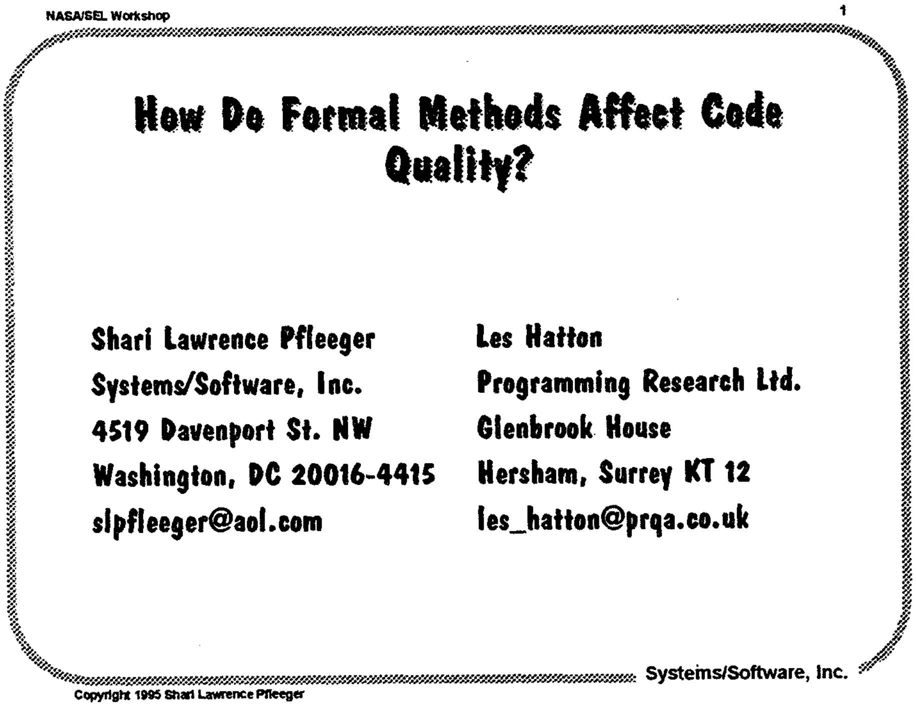
1. In this case study, there is no compelling quantitative evidence that formal design techniques *alone* are responsible for producing code of higher quality than informally-designed code, since the pre-delivery fault profile shows no difference between formally-designed and informally-designed code. On the other hand, the unit testing data show fewer errors revealed in formally-designed code, and post-delivery failures are significantly less for formally-designed code. Thus, formal design together with other techniques have led to code that is highly reliable.

2. Because the high-quality audit profile was uniform and independent of design type, it is likely that it was the formal *specification* that led to components that were relatively simple and independent, making them relatively easy to unit-test.

3. Thorough testing (due to the customer's requirement of 100% statement coverage) combined with thorough specification have resulted in very low failure rates.

4. Even with thorough testing and specification, the CDIS code contained some latent faults that were revealed only through static inspection. Thus, to achieve the highest levels of reliability, developers should combine formal specification and good testing with static inspection.

In other words, formal specification and design are effective under some but not necessarily all circumstances. Their effectiveness may be improved by supplementing them with other approaches, so that in concert they address most of the likely problems of software development. Moreover, formal methods may be more effective in acting as a catalyst for other techniques, especially testing, by virtue of producing testable components.



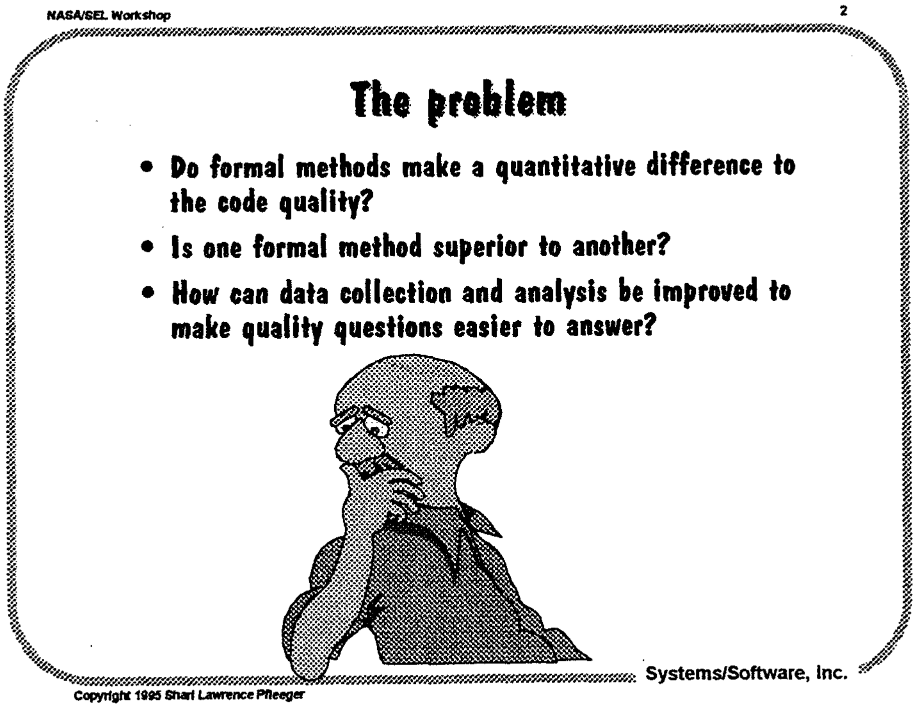
NASA/SEL Workshop 1

How Do Formal Methods Affect Code Quality?

Shari Lawrence Pfleeger
Systems/Software, Inc.
4519 Davenport St. NW
Washington, DC 20016-4415
slpfleeger@aol.com

Les Hatton
Programming Research Ltd.
Glenbrook House
Hersham, Surrey KT 12
les_hatton@prqa.co.uk

Systeims/Software, Inc.



NASA/SEL Workshop 2

The problem

- Do formal methods make a quantitative difference to the code quality?
- Is one formal method superior to another?
- How can data collection and analysis be improved to make quality questions easier to answer?

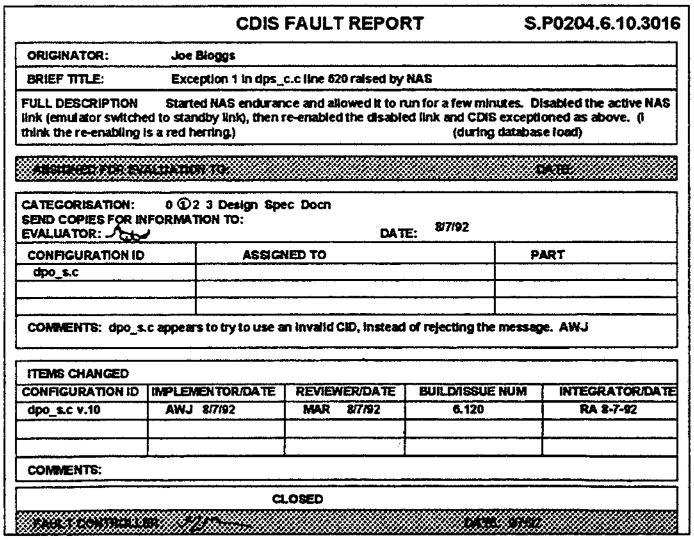
Systems/Software, Inc.

The opportunity

- **Praxis built CDIS using several formal methods**
- **Specification**
  - formal specification of the system (language similar to VDM)
  - user interface specified using prototyping, pictures, text, state transition diagrams
  - concurrency specified using DFDs and Milner's communicating sequential processes (CCS)
- **Design**
  - application code used VDM refinements of core specification
  - user interface designed informally, using pseudocode
  - concurrency defined as FSMs
  - local area network designed using VDM and CCS (some formal proofs done)

Systems/Software, Inc.

The data available



CDIS FAULT REPORT S.P0204.6.10.3016

ORIGINATOR:	Joe Bloggs
BRIEF TITLE:	Exception 1 in dps_c.c line 620 raised by NAS

FULL DESCRIPTION Started NAS endurance and allowed it to run for a few minutes. Disabled the active NAS link (emulator switched to standby link), then re-enabled the disabled link and CDIS exceptioned as above. (I think the re-enabling is a red herring.) (during database load)

ASSIGNED FOR EVALUATION TO: DATE:

CATEGORISATION: 0 [✓ 1] 2 3 Design Spec Docn
SEND COPIES FOR INFORMATION TO:
EVALUATOR: DATE: 8/7/92

CONFIGURATION ID	ASSIGNED TO	PART
dpo_s.c		

COMMENTS: dpo_s.c appears to try to use an invalid CID, instead of rejecting the message. AWJ

ITEMS CHANGED

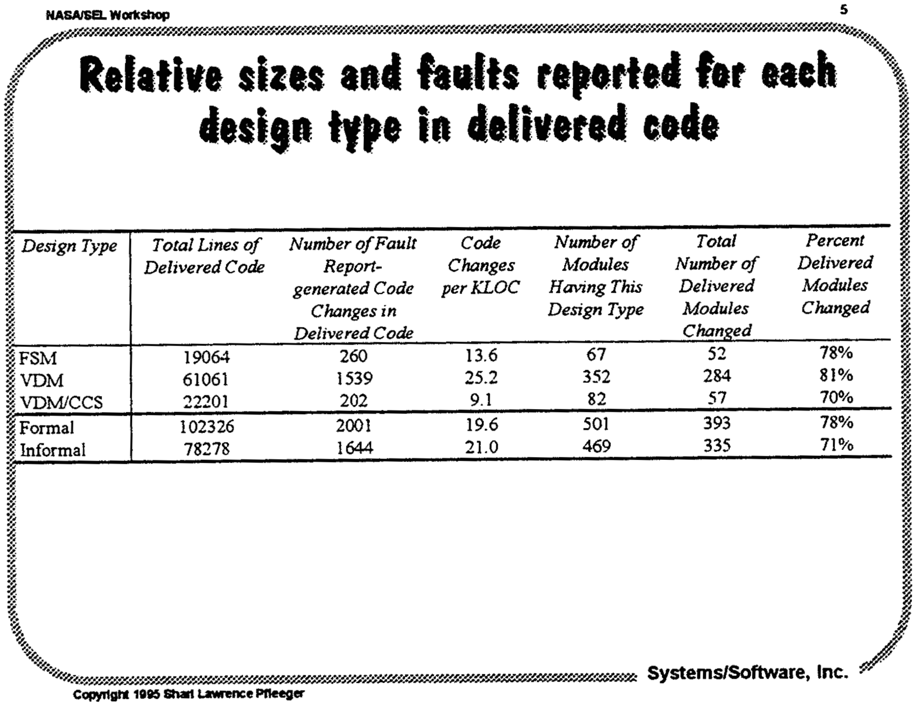
CONFIGURATION ID	IMPLEMENTOR/DATE	REVIEWER/DATE	BUILD/ISSUE NUM	INTEGRATOR/DATE
dpo_s.c v.10	AWJ 8/7/92	MAR 8/7/92	6.120	RA 8-7-92

COMMENTS:

CLOSED

FAULT CONTROLLER: DATE: 8/7/92

Systems/Software, Inc.

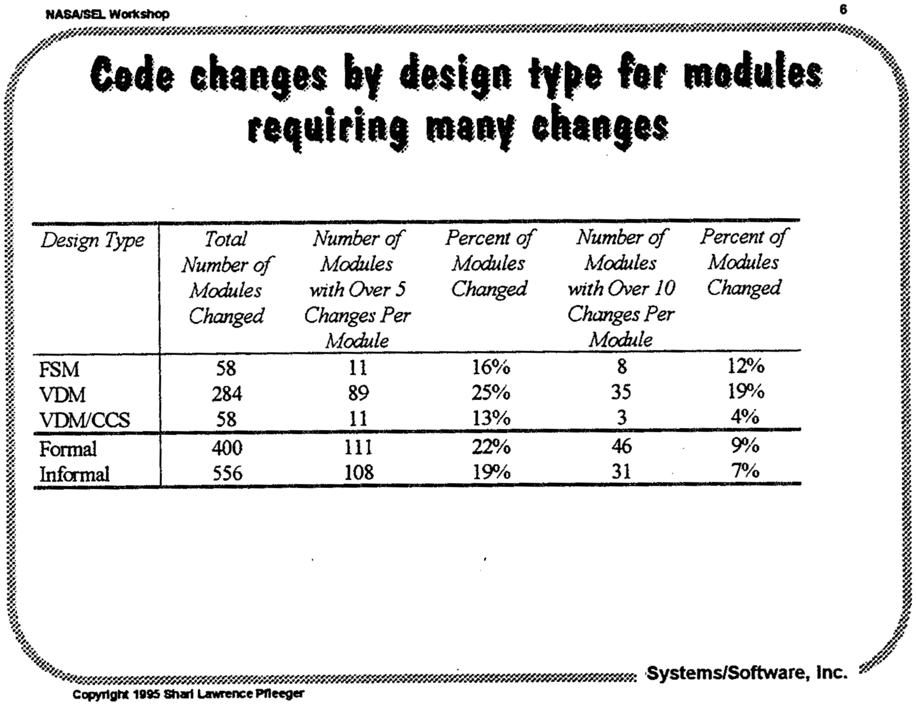


NASA/SEL Workshop 5

Relative sizes and faults reported for each design type in delivered code

Design Type	*Total Lines of Delivered Code*	*Number of Fault Report-generated Code Changes in Delivered Code*	*Code Changes per KLOC*	*Number of Modules Having This Design Type*	*Total Number of Delivered Modules Changed*	*Percent Delivered Modules Changed*
FSM	19064	260	13.6	67	52	78%
VDM	61061	1539	25.2	352	284	81%
VDM/CCS	22201	202	9.1	82	57	70%
Formal	102326	2001	19.6	501	393	78%
Informal	78278	1644	21.0	469	335	71%

Systems/Software, Inc.



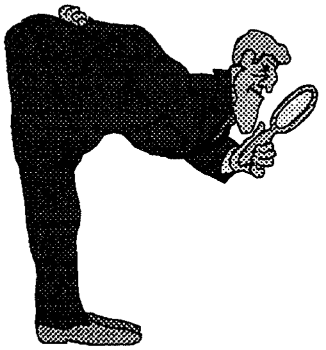
NASA/SEL Workshop 6

Code changes by design type for modules requiring many changes

Design Type	*Total Number of Modules Changed*	*Number of Modules with Over 5 Changes Per Module*	*Percent of Modules Changed*	*Number of Modules with Over 10 Changes Per Module*	*Percent of Modules Changed*
FSM	58	11	16%	8	12%
VDM	284	89	25%	35	19%
VDM/CCS	58	11	13%	3	4%
Formal	400	111	22%	46	9%
Informal	556	108	19%	31	7%

Systems/Software, Inc.

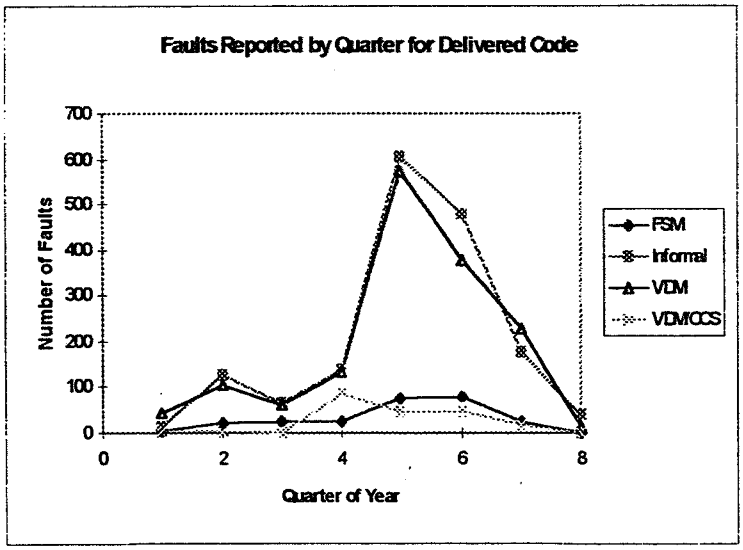
NASA/SEL Workshop 7



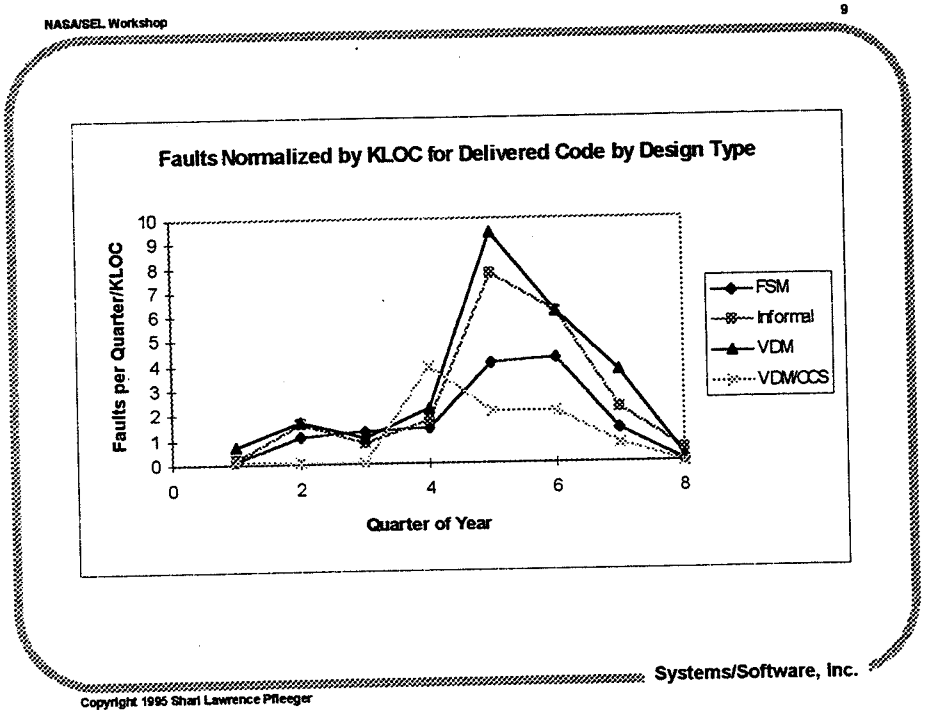
The analysis

Systems/Software, Inc.

NASA/SEL Workshop 8



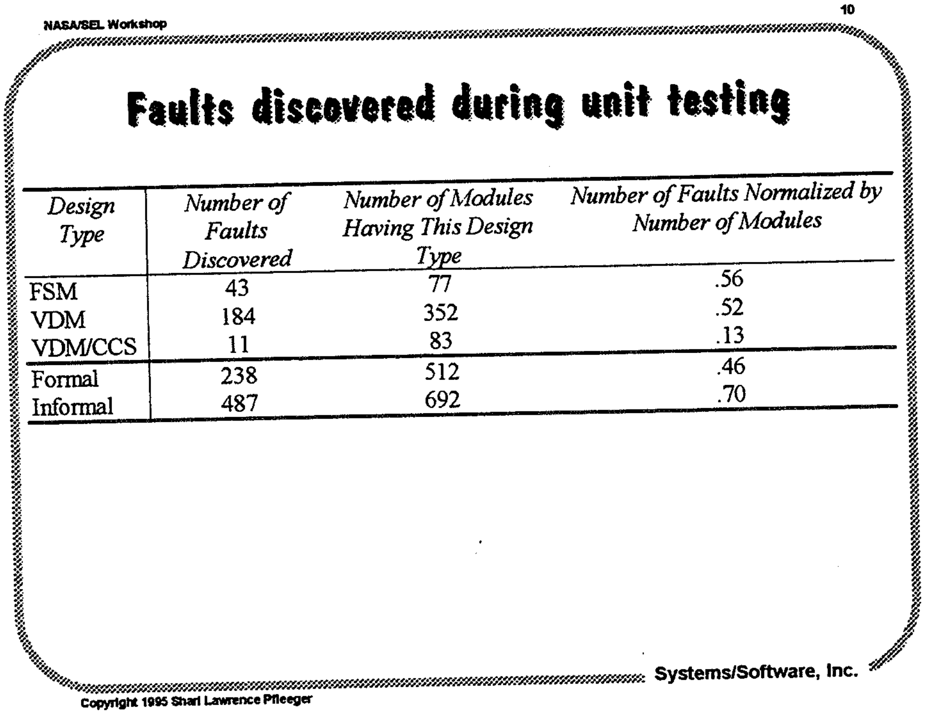
Systems/Software, Inc.



NASA/SEL Workshop 9

Faults Normalized by KLOC for Delivered Code by Design Type

Systems/Software, Inc.

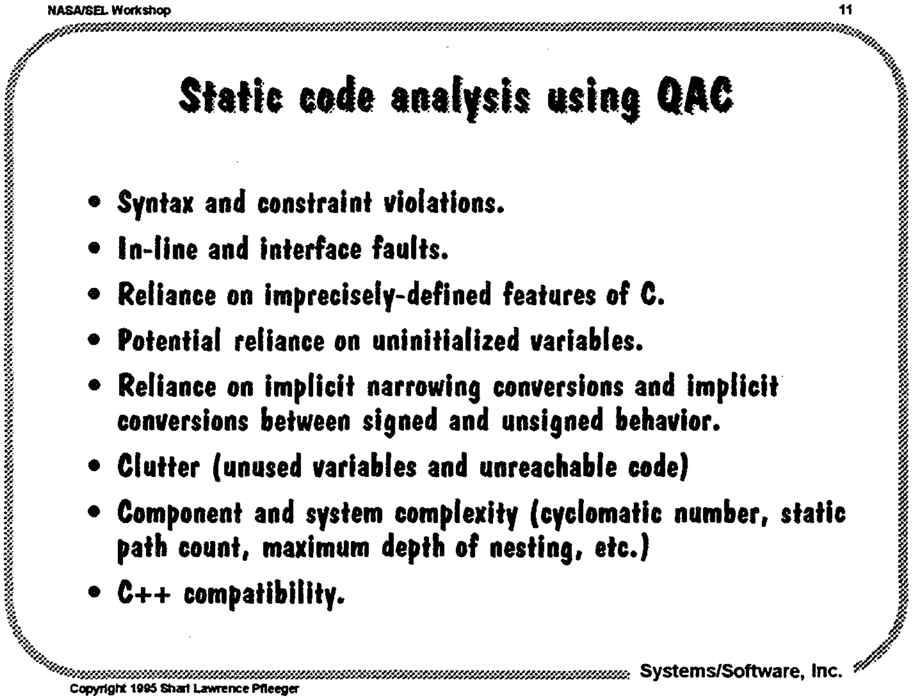


NASA/SEL Workshop 10

Faults discovered during unit testing

Design Type	Number of Faults Discovered	Number of Modules Having This Design Type	Number of Faults Normalized by Number of Modules
FSM	43	77	.56
VDM	184	352	.52
VDM/CCS	11	83	.13
Formal	238	512	.46
Informal	487	692	.70

Systems/Software, Inc.



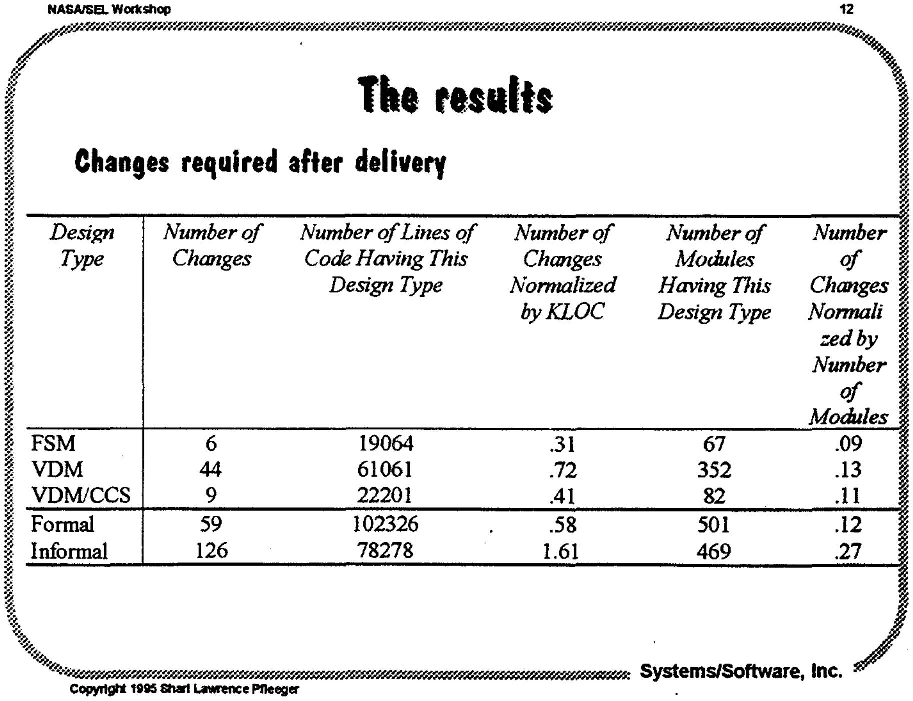
NASA/SEL Workshop 11

Static code analysis using QAC

- Syntax and constraint violations.
- In-line and interface faults.
- Reliance on imprecisely-defined features of C.
- Potential reliance on uninitialized variables.
- Reliance on implicit narrowing conversions and implicit conversions between signed and unsigned behavior.
- Clutter (unused variables and unreachable code)
- Component and system complexity (cyclomatic number, static path count, maximum depth of nesting, etc.)
- C++ compatibility.

Systems/Software, Inc.

Copyright 1995 Shari Lawrence Pfleeger



NASA/SEL Workshop 12

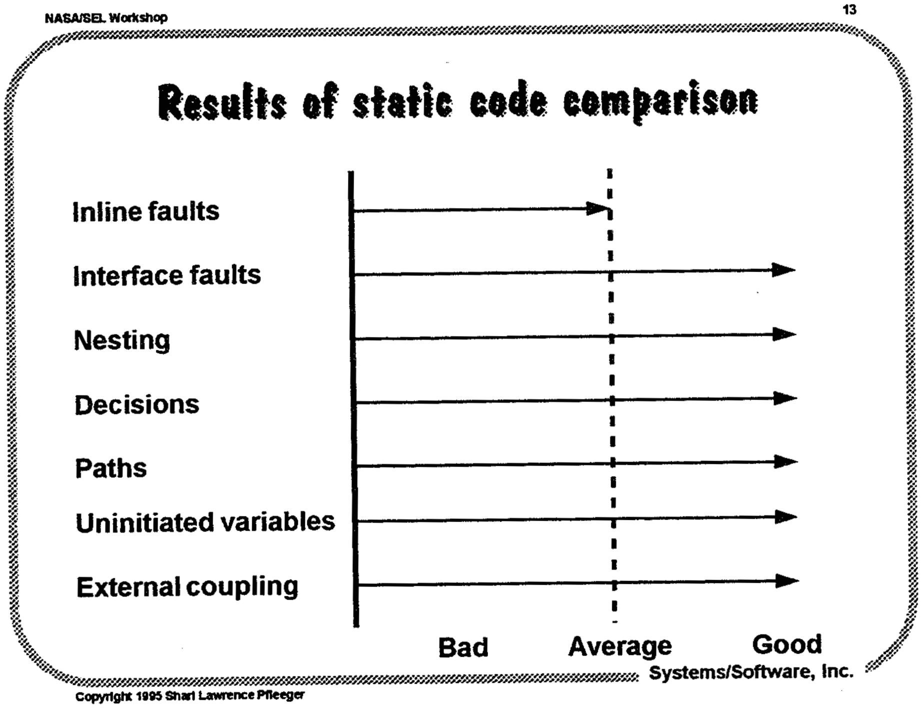
The results

Changes required after delivery

Design Type	*Number of Changes*	*Number of Lines of Code Having This Design Type*	*Number of Changes Normalized by KLOC*	*Number of Modules Having This Design Type*	*Number of Changes Normalized by Number of Modules*
FSM	6	19064	.31	67	.09
VDM	44	61061	.72	352	.13
VDM/CCS	9	22201	.41	82	.11
Formal	59	102326	.58	501	.12
Informal	126	78278	1.61	469	.27

Systems/Software, Inc.

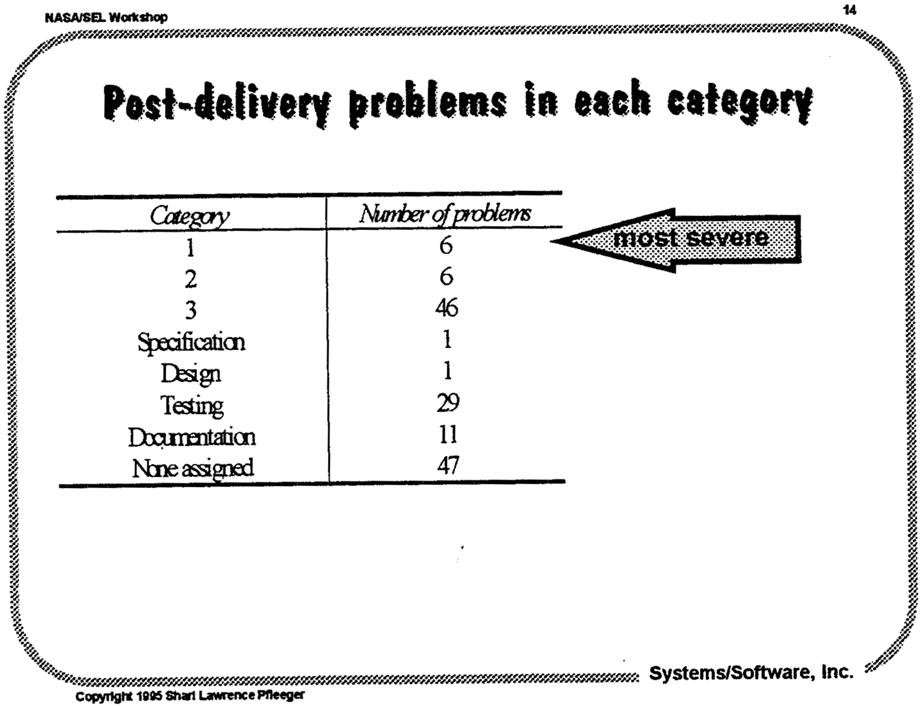
Copyright 1995 Shari Lawrence Pfleeger



NASA/SEL Workshop 13

Results of static code comparison

Systems/Software, Inc.



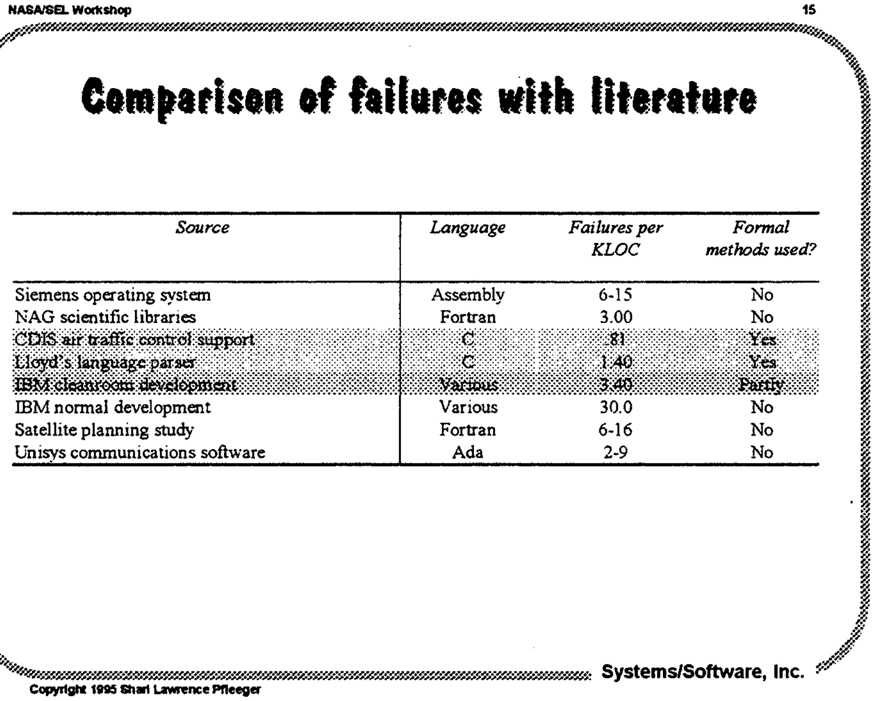
NASA/SEL Workshop 14

Post-delivery problems in each category

Category	Number of problems
1	6
2	6
3	46
Specification	1
Design	1
Testing	29
Documentation	11
None assigned	47

most severe

Systems/Software, Inc.



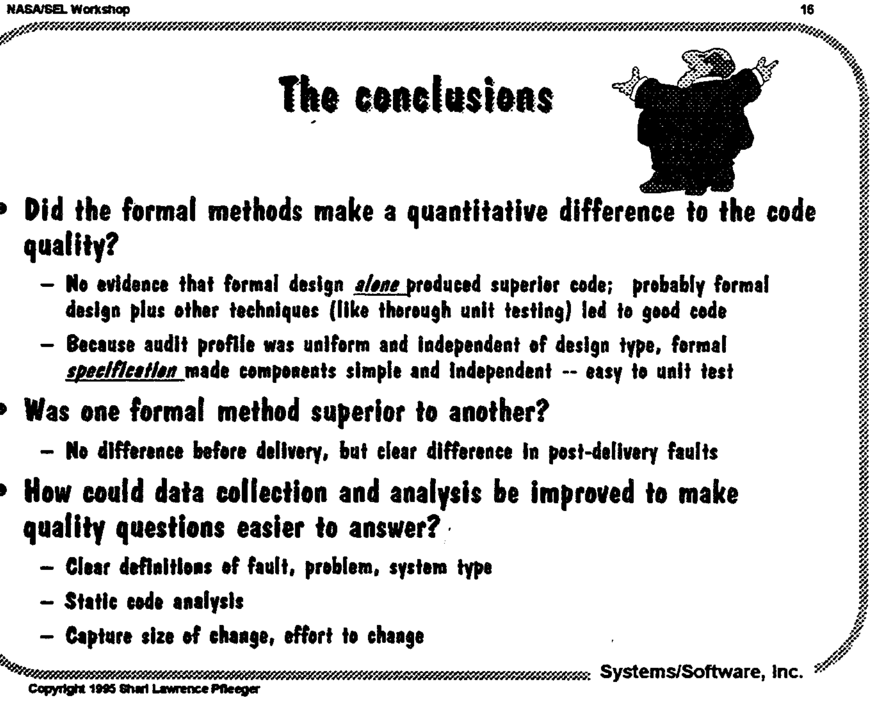
NASA/SEL Workshop 15

Comparison of failures with literature

Source	*Language*	*Failures per KLOC*	*Formal methods used?*
Siemens operating system	Assembly	6-15	No
NAG scientific libraries	Fortran	3.00	No
CDIS air traffic control support	C	.81	Yes
Lloyd's language parser	C	1.40	Yes
IBM cleanroom development	Various	3.40	Partly
IBM normal development	Various	30.0	No
Satellite planning study	Fortran	6-16	No
Unisys communications software	Ada	2-9	No

Systems/Software, Inc.

Copyright 1995 Shari Lawrence Pfleeger



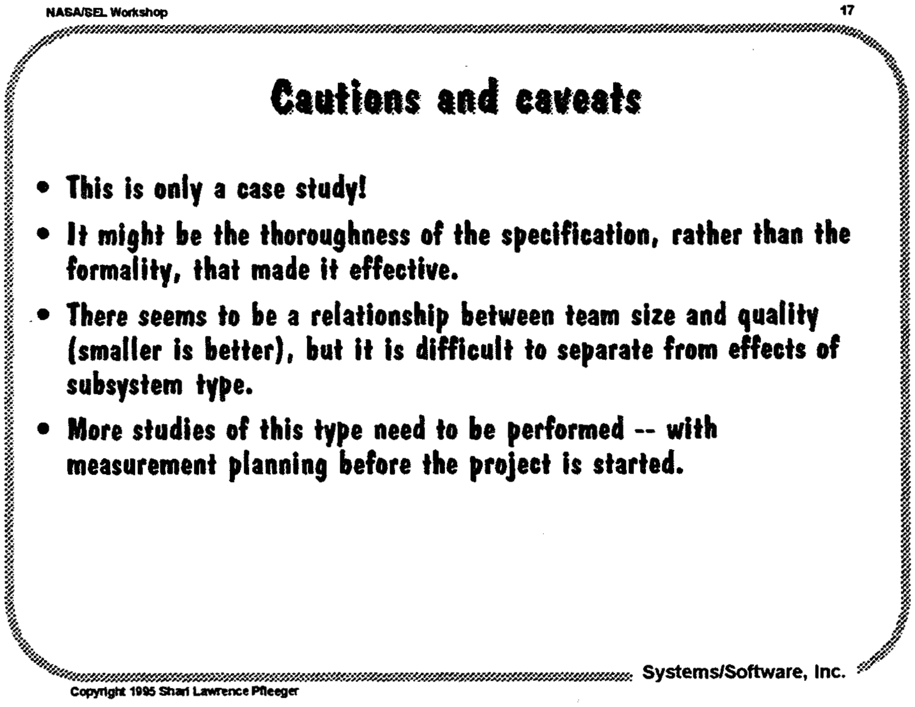
NASA/SEL Workshop 16

The conclusions

- **Did the formal methods make a quantitative difference to the code quality?**
  - No evidence that formal design *alone* produced superior code; probably formal design plus other techniques (like thorough unit testing) led to good code
  - Because audit profile was uniform and independent of design type, formal *specification* made components simple and independent -- easy to unit test
- **Was one formal method superior to another?**
  - No difference before delivery, but clear difference in post-delivery faults
- **How could data collection and analysis be improved to make quality questions easier to answer?**
  - Clear definitions of fault, problem, system type
  - Static code analysis
  - Capture size of change, effort to change

Systems/Software, Inc.

Copyright 1995 Shari Lawrence Pfleeger



NASA/SEL Workshop 17

Cautions and caveats

- This is only a case study!
- It might be the thoroughness of the specification, rather than the formality, that made it effective.
- There seems to be a relationship between team size and quality (smaller is better), but it is difficult to separate from effects of subsystem type.
- More studies of this type need to be performed -- with measurement planning before the project is started.

Systems/Software, Inc.

Panel Discussion: Has the Investment in Process Demonstrated an Impact on Software?

OMIT
THIS
PAGE

Moderator: Vic Basili, University of Maryland

Tom DeMarco, The Atlantic Systems Guild, Inc.

Jim Herbsleb, Software Engineering Institute

Dieter Rombach, University of Kaiserslautern

Tony Wasserman, IDE, Inc.

Omit
TO
END

SEW Panelists Ask, Is Process Enough?

Marueen McSharry
Computer Sciences Corporation

Reprinted with Permission from IEEE Software

Although the question posed at the 20th Annual Software Engineering Workshop was "Has investment in process had an impact on software?" the answers from a panel of software-engineering notables steered the discussion toward art versus engineering, covering people, process, tools, and technology along the way.

In a cunning presentation in which he made erudite sport of everything from his fellow panelists to climatic conditions in the conference auditorium, Tom DeMarco of The Atlantic Systems Guild engaged the crowd with his perspective on investment in process. DeMarco claimed that people — not process — represent the bulk of every organization's investment, and people are the repositories for experience, expertise, and skills that represent potential for future advancement. He said that "companies that think of their people as capital assets and invest in them have had huge impact, and companies that don't have not."

DeMarco was joined by three other panelists. Jim Herbsleb, representing the SEI, gave a thorough and, in his words, "predictable" argument for the efficacy of the Capability Maturity Model. "Maturity matters," he said, and proceeded to document his view, citing "substantial evidence that . . . there is business value for making these kinds of improvements." While Herbsleb essentially pitched the SEI party line in his presentation, during the discussion he was open-minded and flexible on process issues. "Don't slavishly follow a model in a mechanical, thoughtless way," he advised, "the CMM is a tool to help you pursue your business objectives." Herbsleb's attitude went a long way toward dispelling the SEI's image as the enforcer of the KPAs.

MORE THAN METRICS

Dieter Rombach, the reigning "king" of software engineering in Europe and a faculty member at the University of Kaiserslautern, argued the importance of using relevant measurements to transform the "art" of software into the "engineering" of software. He stressed that in today's competitive environment credibility must be proven through measurable business processes. "The goal is to move from an ad hoc process, to a repeatable process, to a demonstrable process," Rombach said. True to his roots, Rombach favored metrics programs and "learning organizations" built on the SEL Experience Factory model to help guide "sensible investment in process." And why not? It works for him. Rombach has recently been endowed by the German government to establish a software-engineering research institute within the Fraunhofer Institute, with, as one attendee quipped, "an operating budget greater than the entire software-engineering research budget of Canada."

Tony Wasserman, of Software through Pictures fame, was expected to represent the tools perspective, but actually illustrate — through pictures, naturally — the folly of elevating process, tools, or any other single element as the exclusive solution to software-engineering issues. "Process helps," Wasserman said, "but process can be thrown off by poor management, poor staff, poor tools, poor working environments."

To illustrate this point, he displayed an image of the title characters from the film "Dumb and Dumber" as he posed the question, "If process is everything, then why not hire these guys to build the product?" Wasserman displayed many film stills during the presentation, giving the impression that he may have struck a product-placement deal with a Hollywood studio. Or perhaps he intended for the stills — inasmuch as they represent the essence of American culture — to underscore another of his points: The importance of culture in the technological universe. Wasserman claimed that process is "dependent on the organizational culture" and the "tendencies of the people."

Following the opening presentations, moderator Victor Basili, University of Maryland — whom DeMarco dubbed "the man who has never had an unpublished thought" — took the floor. Finding the role of neutral moderator too confining, Basili seemed irresistibly drawn to the overhead projector where he weighed in with a quickly sketched slide depicting the relationship of art and engineering in software. Lively debate ensued, especially after what Rombach labeled DeMarco's "second presentation" — in which DeMarco, according to his own characterization, "attacked, ad homonym" each of the panelists and the moderator.

BACK TO REALITY

Although DeMarco dominated the panel presentation, when the discussion was opened to the floor, virtually all of the questions were directed to Herbsleb of the SEI and addressed CMM-related issues - a telling sign that in practice, anyway, whatever its shortcomings, people are grappling with CMM. Herbsleb remained eminently calm as DeMarco beseeched the audience to consider the "very upsetting possibility" that "getting higher and higher CMM levels may well be making us more and more effective at doing things that are less and less worth doing." Attendees asked Herbsleb why the KPAs are organized as they are, why the CMM doesn't provide specific guidance for measurement, and "What if I'm a level-1 and I think a level-3 KPA might be helpful to me? Can I do it?" Herbsleb advised one and all to "do what works" using CMM as a tool, not a "religious text."

AN OUTRAGEOUS IDEA

John Musa, AT&T Bell Laboratories, who spoke in an earlier workshop session, approached the audience microphone and suggested that "Process is something you define when you have no confidence in your people." This remark drew chortles of appreciation from the crowd. "Dumb and Dumber" redux? While the panelists disagreed with Musa's admittedly "outrageous" statement, his comment spurred the discussion on to a consideration of what combination of elements is needed to effectively engineer software. In the end, each speaker gave his last word, and, not surprisingly, their ideas converged harmoniously in the view that no one element is dominant. People are indeed critical to the software process; "investment" represents not only the training of those people, but the acquisition and application of appropriate tools and technology to support them. Process tools, such as CMM, help people work more effectively. Structured measurement and analysis, such as the Experience Factory model, deliver a credible, justifiable business case for particular process choices.

HARMONY RESTORED

As for the art versus engineering debate, the panel concluded that repeatable processes, far from enslaving us on an assembly line, rather release us from mundane software chores. They free us to apply our creativity and intellectual power to ever more challenging problems: to the "art" of software engineering. Basili's late-breaking slide captured this balance in a curve, rising infinitely over time, in which today's artistic challenges become tomorrow's engineered.

Appendix A: Workshop Attendees

Appendix A: Workshop Attendees

Agresti, Bill W, MITRE Corp
Allen, Charles J., Unisys Corp
Anderson, Barbara, Jet Propulsion Lab
Angevine, Jim, ALTA Systems, Inc
Ayers, Everett, ARINC Research Corp

Bacon, Beverly, Computer Sciences Corp
Bailey, John, Software Metrics, Inc
Basili, Victor R., University of Maryland
Bassman, Mitchell J., Computer Sciences Corp
Baxter, Mary E., Hughes-STX
Beall, Shelly, Social Security Administration
Becker, Shirley, The American University
Beifeld, David, Unisys Corp
Bennett, Stephen, Joint Warfare Analysis Center
Bernier, Cathy, BTG, Inc
Berthiaume, Fred, Unisys Corp
Beswick, Charles A., Jet Propulsion Lab
Bhatia, Kiran, MITRE Corp
Blagmon, Lowell E., Naval Center For Cost Analysis
Bobowiec, Paul W., COMPTEK Federal Systems
Boland, Dillard, Computer Sciences Corp
Bozoki, Martin J., Lockheed Missiles & Space Co, Inc
Briand, Lionel, CRIM
Bristow, John, NASA/GSFC
Brock, O. K., EG&G
Brown, Cindy, Computer Sciences Corp
Bryant, Joan L., IRS

Calavaro, Giuseppe F., Hughes Information Technology Corp
Caldiera, Gianluigi, University of Maryland
Callahan, Jack, West Virginia University
Carlson, Randall, NSWCDD
Carnahan, Rich, Lockheed Martin
Childers, Donald E., Joint Warfare Analysis Center
Christensen, Joel A., TASC
Chu, Martha, Computer Sciences Corp
Chu, Richard, Loral AeroSys
Church, Vic, Computer Sciences Corp
Condon, Steven E., Computer Sciences Corp
Corbin, Regina, Social Security Administration
Corderman, Elizabeth, NASA/GSFC
Cortes, Elvia E., DISA
Crawford, Art, Mantech Advanced Technology Systems
Cuesta, Ernesto, Computer Sciences Corp
Cummins, Mary, Loral AeroSys
Cusick, James, AT&T Bell Labs

D'Agostino, Jeff, The Hammers Co
Daku, Walter, Unisys Corp
Daniele, Carl J., NASA/LeRC
DeMarco, Tom, The Atlantic Systems Guild
Debaud, Jean-Marc, Georgia Institute of Technology
Decker, William J., Computer Sciences Corp
Deutsch, Michael S., Hughes Applied Info Systems, Inc
Deyton, Ray, Loral AeroSys
DiNunno, Donn, Computer Sciences Corp
Diskin, David H., Defense Information Systems Agency
Doland, Jerry T., Computer Sciences Corp
Dolphin, Leonard, ALTA Systems, Inc
Donahue, Brian E., Harris Corp - GASD
Drappa, Anke, Universitat Stuttgart
Dudash, Ed, Naval Surface Warfare Center
Duffy, Terry, McDonnell Douglas

Easterbrook, Steve, West Virginia University
Edelson, Robert, Jet Propulsion Lab
Elliott, Frank, Social Security Administration
Estep, James, NASA IV&V Facility

Fagerhus, Geir, Q-Labs
Farr, William H., Naval Surface Warfare Center
Farrukh, Ghulam, George Mason University
Feerrar, Wallace, MITRE Corp
Fernandes, Vernon, Computer Sciences Corp
Field, Brian S., Computer Sciences Corp ISMD
Fike, Sherri, Ball Aerospace
Fitch, Chris, IIT Research Institute
Fleming, Barbara, DISA
Flora, Jackie, Social Security Administration
Forsythe, Ron, NASA/Wallops Flight Facility
Frahm, Mary J., Computer Sciences Corp
Funch, Paul G., MITRE Corp
Futcher, Joseph M, Naval Surface Warfare Center

Gant, Donna, GDE Systems, Inc
Gantzer, D. J., Loral Federal Systems
Geiger, Jennifer, NASA/GSFC

Gieser, Jim, Unisys Corp
Godfrey, Sally, NASA/GSFC
Golden, John R., Information Technologies
Goodman, Nancy, NASA/GSFC
Green, David S., Computer Sciences Corp
Green, Scott, NASA/GSFC
Gregory, Judith A., NASA/MSFC
Griffin, Wesley
Groveman, Brian S., Computer Sciences Corp

Hall, Dana L., SAIC
Hall, Joseph F., DPRO-Westinghouse, ES
Hall, Ken R., Computer Sciences Corp
Halterman, Karen, NASA/GSFC
Harris, Barbara A., USDA
Harris, Chi Cha, DPRO-Westinghouse
Harris, Karlette, PRC, Inc
Heasty, Richard, Computer Sciences Corp
Heintzelman, Clinton L., US Air Force
Heller, Gerry H., Computer Sciences Corp
Herbsleb, Jim, Software Engineering Institute
Hill, Ken, Unisys Corp
Hirsch, Steven J., DoD
Hoffmann, Kenneth, Ryan Computer Systems, Inc
Holmes, Joseph A., IRS
Hopkins, Harry A., DISA

Janzon, Tove, Q-Labs
Jay, Elizabeth M., NASA/GSFC
Jefferson, Karen, Computer Sciences Corp
Jeletic, Jim, NASA/GSFC
Jeletic, Kellyann, NASA/GSFC
Jones, Christopher C., IIT Research Institute
Jones, Jay, OAO Corp
Jordan, Gary, Unisys Corp
Jordano, Tony J., SAIC
Joyce, Keith, GDE Systems, Inc

Kalin, Jay, Loral AeroSys
Kanzinger, Erica, Health Care Financing Administration
Kassebaum, Robert, MCI
Kelly, John C., Jet Propulsion Lab
Kemp, Kathryn M., Office of Safety & Mission Assurance
Kester, Rush W., Computer Sciences Corp
Kettenring, Jon R., BELLCORE
Kim, Yong-Mi, University of Maryland
Kontio, Jyrki, University of Maryland
Kraft, Steve, NASA/GSFC
Kramer, Nancy, DynCorp/Viar
Kronisch, Mark, US Census Bureau
Kuhne, Fran, Social Security Administration
Kurihara, Tom M., Logicon, Inc
Lakhotia, Arun, University of Southwestern Louisiana

Landis, Linda C., Computer Sciences Corp
Landry, Huet C., DISA
Lane, Allan C., AlliedSignal Technical Services Corp
Lang, Keith R., Lockheed Martin
Lanubile, Filippo, University of Maryland
Lawrence Pfleeger, Shari, Systems/Software, Inc
Leach, Ronald J., Howard University
Leasure, William C., DPRO-Westinghouse
Leydorf, Steven M., IIT Research Institute
Liebermann, Roxanne, US Census Bureau
Liu, Jean C., Computer Sciences Corp
Livingston, Karen, IIT Research Institute
Loesh, Bob E., Software Engineering Sciences, Inc
Lucas, Janice P., US Treasury Department
Luczak, Ray W., Computer Sciences Corp
Ludford, Joseph F., Computer Sciences Corp
Lyons, Howie, AFPCA/GADB

Mahajan, Rohit, University of Maryland
Marciniak, John J., Kaman Sciences Corp
Martinez, Bill, Loral Federal Systems
Maury, Jesse, Omitron, Inc
McCafferty, Brian, XonTech, Inc
McGarry, Frank E., Computer Sciences Corp
McGarry, Mary Ann, IIT Research Institute
McGrane, Janet K., US Census Bureau
McGuire, Gene, American University
McRoberts, Terry L., NASA/GSFC
McSharry, Maureen, Computer Sciences Corp
Melo, Walcelio L., University of Maryland
Michel, Del, DISA/JIEO/CES/JEBEC
Mike, Debbie, Unisys Corp
Miller, Martin E., DPRO-Westinghouse
Morris, Jeff, Lockheed Martin Corp
Mucci, David, Computer Sciences Corp
Mulvaney, Bill, DISA
Musa, John D., AT&T Bell Labs
Myers, Philip I., Computer Sciences Corp

Narrow, Bernie, AlliedSignal Technical Services Corp
Nero, Theresa
Nestlerode, Howard, Unisys Corp
New, Ronald, COMPU-HELP
Nokovich, Sandra L., US Census Bureau
Norcio, Tony F., University of Maryland-Baltimore Co

Obenzu, Ray, Software Engineering Institute

Page, Gerald T., Computer Sciences Corp
Pajerski, Rose, NASA/GSFC
Paletar, Teresa L., Naval Air Warfare Center
Panlilio-Yap, Nikki M., Loral Software Systems Resource Center
Parra, Amy T., Computer Sciences Corp
Patel, Shirishbhai, NASA/KSC
Pavnica, Paul, Treasury/Fincen
Pedersen, Bonnie, Loral AeroSys
Peeples, Ron L., Intermetrics
Perry, Howard, Computer Sciences Corp
Pfitzer, Bonita, Madison Research Corp
Poe, Kevin, NASA IV&V Facility
Pollack, Ida, Cybersoft
Pollack, Jay, Computer Sciences Corp
Polly, Mike, Raytheon
Powers, Larry T., Unisys Corp
Pugliese, Tom, QSS Group

Quann, Eileen S., Fastrak Training, Inc

Reesman, Leslie J., Hernandez Engineering, Inc
Regardie, Myrna L., Computer Sciences Corp
Rizer, Stephani, NAWC-AD
Rodgers, Thomas M., Lockheed-Martin
Rombach, H. Dieter, University of Kaiserslautern
Rosenberg, Linda H., Unisys Corp
Russell, Earl, USDA

Sabolish, George J., NASA IV&V Facility
Samuels, George, Social Security Administration
Sanderson, Peter, Southwest Missouri State University
Sands, Judy, US Census Bureau
Sands, Robert D., US Census Bureau
Santiago, Richard, Jet Propulsion Lab
Sauble, George, Omitron, Inc
Schappelle, Sam, IBM Object Technology University
Schneider, Laurie, General Sciences Corp/SAIC
Schneidewind, Norman F., Naval Postgraduate School
Schuler, Pat M., NASA/LaRC
Schulterbrandt, Sherwin D., Health Care Financing Administration
Schwartz, Benjamin L., Consultant
Schweiss, Robert, NASA/GSFC
Seaman, Carolyn B., University of Maryland
Seaton, Bonita, NASA/GSFC
Seidewitz, Ed, NASA/GSFC
Selmon, Lisa, SAIC/ASSET
Shah, Rohini, US Census Bureau
Sheckler, John D., AlliedSignal Technical Services Corp
Sheffler, John D., MITRE Corp
Shneiderman, Ben, University of Maryland
Shull, Forrest, University of Maryland
Sim, Ed, Bowie State University
Singer, Carl A., BELLCORE
Slade, Lucius, USDA
Slonim, Jacob, IBM Canada Lab - CAS
Smidts, Carol, University of Maryland
Smith, Donald, NASA/GSFC
Smith, George F., Space & Naval Warfare Systems Command
Smith, Vivian A., FAA
Solomon, Arthur, DISA
Sorumgard, Sivert, University of Maryland
Sova, Don, NASA/HQ
Spangler, Alan, IBM
Sparmo, Joe, NASA/GSFC
Squires, Burton E., Orion Scientific, Inc
Stagmer, Cheryl A., Health Care Financing Administration
Stankewicz, Bonnie, Computer Sciences Corp
Stark, Michael, NASA/GSFC
Stauffer, Michael, Lockheed Martin, M&DSO
Steinberg, Sandee, Computer Sciences Corp
Stoddard, Robert W., Texas Instruments
Stoos, Pat, Social Security Administration
Sukri, Judin, Computer Sciences Corp
Swain, Barbara, University of Maryland
Szulewski, Paul A., CS Draper Labs, Inc

Tasaki, Keiji, NASA/GSFC
Taylor, Michael, Computer Sciences Corp
Tesoriero, Roseanne, University of Maryland
Thomas, William, MITRE Corp
Thomason, Clarke, Pailen-Johnson Associates, Inc
Trachta, Greg, Unisys Corp
Tran, Tuyet-Lan, Jet Propulsion Lab
Tupman, Jack R., Jet Propulsion Lab

Ulery, Bradford T., MITRE Corp
Underwood, David A., IIT Research Institute

Valett, Jon, NASA/GSFC
Valett, Susan, NASA/GSFC
Van Hee, Lee, Boeing
Veillon, Nancy, Social Security Administration
Voigt, Susan J., NASA/LaRC

Wackley, Joseph A., Jet Propulsion Lab
Waligora, Sharon R., Computer Sciences Corp
Walker, James A., DoD
Walsh, Chip, IRS
Walsh, Chuck, RMS Associates

Wasserman, Tony, IDE
Waszkiewicz, Mary Lily, Computer Sciences Corp
Watson, Jim, NASA/LaRC
Weisenberger, Mike, McDonnell Douglas
Weiss, Sandy L., IIT Research Institute
West, Tim, DISA
Weszka, Joan, Loral Federal Systems Group
Wetherholt, Martha S., NASA/LeRC
Whisenand, Tom, Social Security Administration
Whitehead, John W., COMPTEK Federal Systems
Widmaier, James, DoD
Williams, James K, DISA
Wilson, Robert K., Jet Propulsion Lab
Wohlin, Claes, Lund University, Sweden
Wolf, Bryan, McDonnell Douglas
Wolfish, Hannah K., Social Security Administration
Wood, Richard J., Computer Sciences Corp
Woodyard, Charles E., NASA/GSFC

Young, Andy, Young Engineering Services, Inc
Young, Wendall, Pailen-Johnson Associates, Inc

Zeitvogel, Barney, SECON
Zelkowitz, Marv, University of Maryland
Zimet, Beth, EIS

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.1McGarry, 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-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. 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 and Object-Oriented Design 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-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., 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., 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

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. Melo, C. Seaman, and V. 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

[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. 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 Insitute, 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

[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

[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

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.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1995

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
Software Engineering Laboratory Series
Proceedings of the 20th 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-95-004

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—1998–208616

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category: 82
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
380

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