19990025789

NASA/TM-1998-208618

957427

I

Proceedings of the Twenty-Second Annual Software Engineering Workshop

December 3–4, 1997

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, Flight Dynamics Systems Branch

The University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Development and Software Engineering Organization

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

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

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

or by writing to:

Systems Integration and Engineering Branch
Code 581
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)

Page

Session 1: The Software Engineering Laboratory

The SEL Adapts to Meet Changing Times
R. Pajerski, NASA/Goddard, and V. Basili, University of Maryland

The Package-Based Development Process in the FDD
A. Parra, Computer Sciences Corporation, C. Seaman and V. Basili, University of Maryland,
S. Kraft, NASA/Goddard, S. Condon, and S. Burke, Computer Sciences Corporation,
D. Yakimovich, University of Maryland

The Web Measurement Environment (WebME):
A Tool for Combining and Modeling Distributed Data
R. Tesoriero and M. Zelkowitz, University of Maryland

360774

The SEL Adapts to Meet Changing Times

Rose S. Pajerski
NASA, Goddard Space Flight Center

Flight Dynamics Division
Greenbelt, MD 20771
(301) 286-3010
rose.pajerski@gsfc.nasa.gov

Victor R. Basili
Computer Science Department/
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
(301) 405-2668
basili@cs.umd.edu

Background

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

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

Characteristics	Applications	
	Ground Systems	Simulators
System Size	150 - 400 KSLOC	40 - 80 KSLOC
Project Duration	1.0 – 2.0 years	.5 – 1.0 years
Staffing (technical)	4-10 staff-years	1 – 2 staff-years
Language	C, C++, FORTRAN	Ada, FORTRAN
Hardware	Workstations	Workstations

Table 1. Characteristics of SEL Projects

During the past 20+ years, the SEL's overall goal has remained the same: to improve the FDD's software products and processes in a measured manner. This requires that each development and maintenance effort be viewed, in part, as a SEL experiment, which 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.

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

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. Only then can new technologies be introduced and assessed (phase two) with regard to both process changes and product impacts. There are typically several studies ongoing at any one time, which take 1-3 years to complete. The third phase synthesizes the results of the first two phases into various packages such as process tailoring guidance, training materials, and tools and guidebooks. These results are then fed back into the cycle for subsequent projects to use and benefit from.

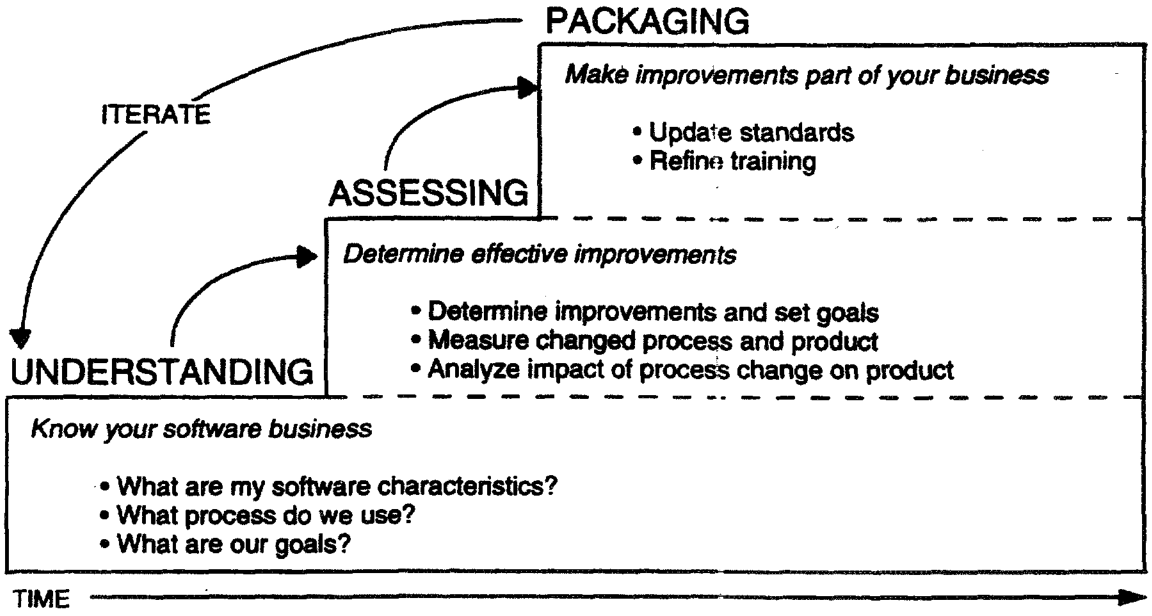

PACKAGING
ITERATE
Make improvements part of your business
• Update standards
• Refine training
ASSESSING
Determine effective improvements
• Determine improvements and set goals
• Measure changed process and product
• Analyze impact of process change on product
UNDERSTANDING
Know your software business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

Figure 1. SEL Process Improvement Paradigm

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

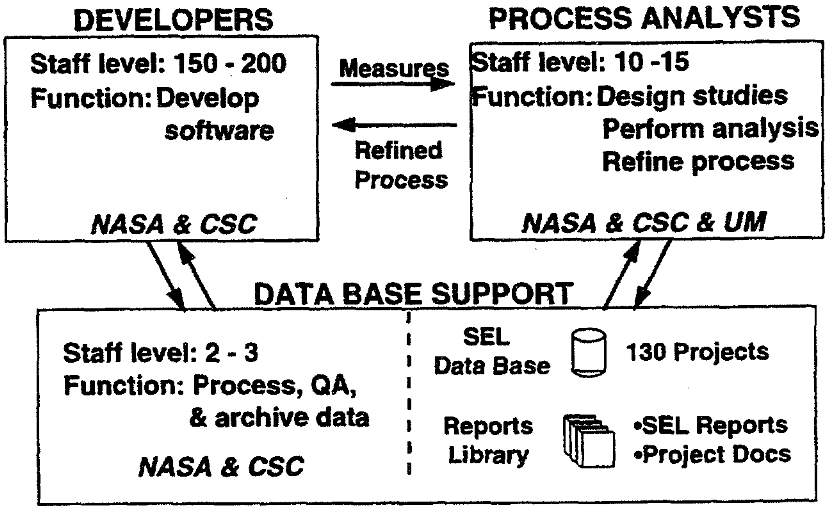


Figure 2. SEL Organizational Structure

Drivers for Change

The SEL has faced a number of changes over the past few years brought about by both environmental and technical factors. These factors include the phase-out of the mainframe systems and subsequent transition to workstations, growth of object-orientated languages (Ada, and C++), and the increasing usage of Commercial-off-the shelf (COTS) scientific application products. However, the latest challenge goes deeper, to a more fundamental, organizational level.

The drivers for this start at the NASA-wide level and extend throughout both Goddard and the local division organizations. The 1997 NASA Strategic Plan has several elements that impact Goddard structures:

- The Enterprise Organizations, such as Mission to Planet Earth and Human Exploration of Space, will become a source of direct funding for SEL studies. This implies a more involved customer who will expect the SEL to be able to show some cost benefit fairly quickly.
- Mission managers will need performance data for both in-house as well as acquired software efforts to show schedule and budget conformance.
- The NASA Software Strategic Plan defines specific goals for software management, assurance, and improvement organizations to attain that may impact SEL activities.

Other drivers arise from the Goddard Strategic Plan and Goddard's new organizational structure. While teams have always provided support for missions, the roles and responsibilities of teams have been expanded. A mission team will now support a specific project throughout all mission phases, thereby involving development organizations earlier in the project. The scope of the development organization has also been broadened to include end-to-end information systems (ground and onboard). The new functionally based Information Systems Center (ISC) shown in Figure 3 will be the focal point for software expertise and a systems support infrastructure throughout Goddard.

Information Systems Center (ISC)

Flight dynamics
Science data processing
Science data visualization and analysis
Command, control, integration and test
Ground and space network
Flight and embedded software
Information system technology

Figure 3. ISC Functions

Given these new strategic and organizational mandates, the SEL has an opportunity to leverage its capabilities to help meet the ISC's expanded responsibilities in several areas:

- Build an improvement organization within the ISC that will increase the competency of its software professionals, thereby increasing the quality of Goddard software systems
- Model and characterize software systems in use on the ground and onboard spacecraft
- Transfer and help tailor proven development and maintenance technologies to new domains, internal and external to GSFC.

Organizational Adaptation

Under the proposed expanded SEL structure, the development organization would expand to include the entire project team while the software engineering process analysts and data base support functions would remain the same (Figure x). However, the scope of work of the process analysts would encompass the end-to-end systems development process, from requirement definition through maintenance and operations. The corresponding metrics used would also change to reflect the additional phases of the system lifecycle under analysis. At a minimum, measures relevant to the requirement definition and operations/delivery processes would need to be included.

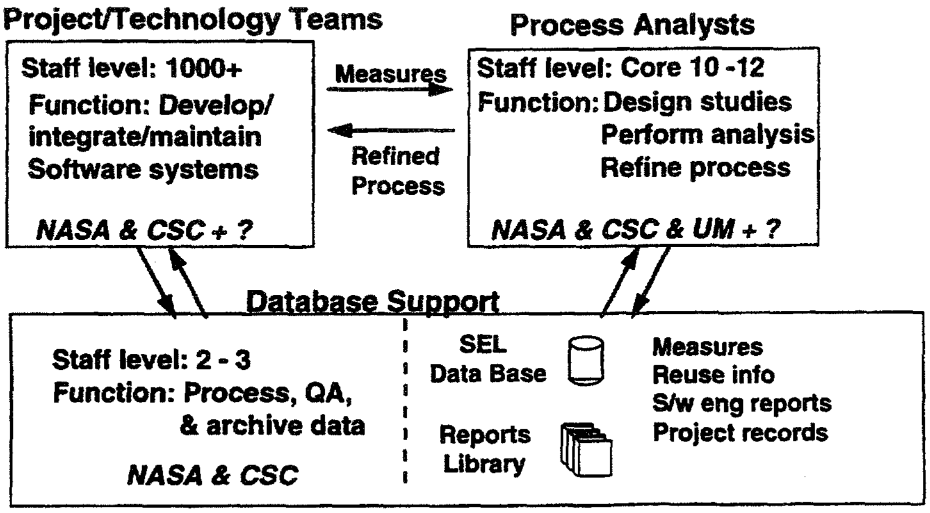


Figure 4. SEL Structure under the ISC

Another organizational issue that will need to be considered is the approach taken in planning SEL work. Over the past several years, the SEL has been managed by a group of 4-6 senior managers who would meet 2-3 times a year to set local improvement goals. Based on these goals, members of the group would propose applied research and study areas for the next 1–2 years. Resource requirements would be discussed and teams formed from the three partner organizations.

With the expansion of the SEL's role in the ISC, planning input for the SEL's activities would be solicited from a broader user community across Goddard including project offices. In their yearly planning, SEL managers would respond to an analysis of these project needs in setting improvement goals, selecting study areas and associated metrics, and feeding back results to all involved. The scope of SEL leadership might also grow to encompass other academic and industry partners; however, this aspect needs further study.

Experimental Adaptation

The SEL has conducted hundreds of process technology studies of different size and duration. Some have been multi-year, multi-application studies (e.g., Cleanroom, Ada) while others have been much smaller and quicker (e.g., testing approach). [References 2,3,4] Over time, this has resulted in refinements to the experimental approach itself in two areas: study selection and approach, and types of analysis performed.

The trend has been to perform smaller studies that build upon one another over time. This has two benefits: quicker feedback to the development groups of useful results, and quicker realization of benefits that then accumulate over time. The independent test team study and the ongoing COTS process study are two examples of this. Figure 5 shows the current approach that emphasizes early deployment of new elements of the process to the development groups as the study proceeds. The impact of these changes on the overall process and product can then be demonstrated and incorporated into the organizational baseline, thereby increasing ISC's competitive position.

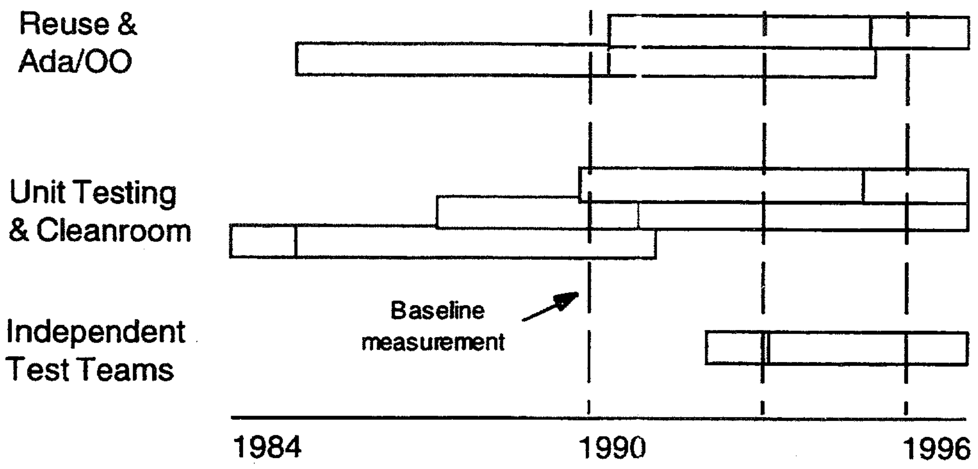


Figure 5. Improvement Cycle Timelines

This emphasis on accelerating analysis has also modified the types of analysis performed. The use of qualitative techniques, such as focused interviews, has increased. The development teams are interviewed at the experiment's start to ensure that their goals and perspectives are factored into the experiment. This domain "discovery " step enhances the SEL process analysts' understanding and facilitates communication throughout the study. Another helpful communication mechanism is the use of online feedback reports to replace face-to-face meetings. As more groups become involved in SEL studies, their use is expected to increase – to become an important technology transfer mechanism also.

Technology Transfer Adaptation

SEL experience with transferring our process technologies and experimental results to the "outside" was predicated on whether the receiving organization was internal to Goddard or external. Internal transfers were supported with a very hands-on approach that included training, tailoring, and impact analysis. However, for external transfers, the users were provided with detailed guides along with some tailoring information but were essentially responsible for implementation and change analysis.

As our understanding of the considerable resources required to be successful at technology transfer has increased, the SEL approach to technology transfer has become more sophisticated. We are developing domain-based techniques to replace the previous "one size fits all" approach. The resulting mechanism considers a range of factors in order to predict the success of a transfer based on key similarities in organization and environment between the organizations involved. The filter can also be used to more easily tailor a particular technology.

Next Steps

Based on the above discussion, there are several steps for the SEL and the ISC to pursue in concert:

1. Profile the ISC organization and establish a new baseline of products developed and processes used in systems development, integration, and maintenance. An understanding of the new operation of the organization is crucial to establish priorities and successfully plan SEL support for them.

2. Select a few projects for focused SEL support – projects tha differ in scope (development vs. COTS integration) and organization (in-house vs. contract). This would involve establishing basic measurement mechanisms as well as feedback and reporting procedures for a subset of new or ongoing projects.

3 Evolve the direction of the SEL to include Enterprise representation and new models for leadership. A key ingredient to the SEL's past success has been the close cooperation between development and process analysis groups – and it will be a challenge to replicate this with other groups, perhaps across different companies.

Expanding the scope and support activities of the SEL will not be easy; however, it will position the ISC to be able to improve Goddard's future systems development efforts.

References

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

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

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

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

The SEL Adapts to Meet Changing Times

Rose Pajerski, NASA GSFC
Victor Basili, University of Maryland

Presentation Outline

- Drivers for Change
- Organizational Adaptation
- Experimentation Adaptation
- Technology Transfer Adaptation
- Next Steps

Drivers

- NASA Strategic Plan
  - Enterprises as involved customer/funding source
  - Performance measurement (Inhouse/Acquired)
  - Software Strategic Plan (Manage/Assure/Improve)
- Strategic Goals of Project Goddard
  - Expanded team structure across all mission phases
  - Focus on end-to-end information systems
  - ISO 9001 certification
- New Organizational Structure Proposed
  - Software responsibility & expertise in single organization
  - Significant infrastructure assets

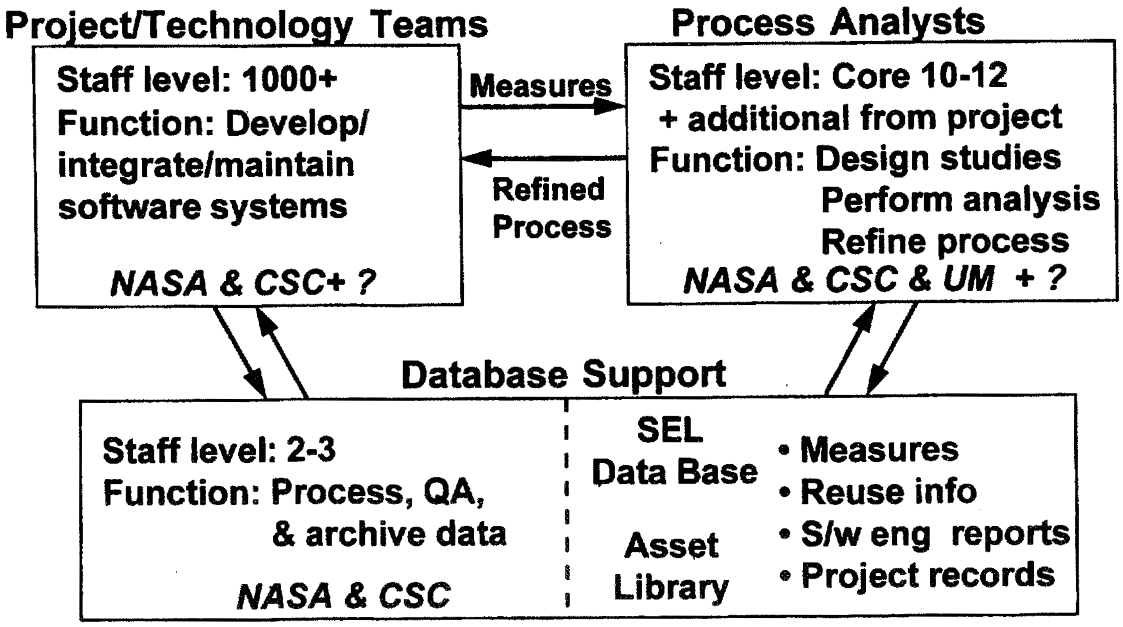
Leveraging SEL Experience

- Building an improvement organization
  - Understand/characterize new domain elements
  - Use SEL infrastructure (approach/measurement)
- Studying end-to-end software systems
  - Model effort/ schedule/ errors
  - Develop reuse assets (process & product)
- Transferring technology based on domain characteristics
  - Internal and external transfers
  - Tailor development, integration & maintenance practices

Organizational Adaptation

- ■ Drivers for Change
- ■ Organizational Adaptation
  - ◆ SEL Structure
  - ◆ Work Planning Approach
  - ◆ SEL Infrastructure
- ■ Experimentation Adaptation
- ■ Technology Transfer Adaptation
- ■ Next Steps

SEL Structure (Near-term Concept)



Project/Technology Teams

Staff level: 1000+
Function: Develop/ integrate/maintain software systems

NASA & CSC+ ?

Measures

Refined Process

Process Analysts

Staff level: Core 10-12
+ additional from project
Function: Design studies
Perform analysis
Refine process

NASA & CSC & UM + ?

Database Support

Staff level: 2-3
Function: Process, QA, & archive data

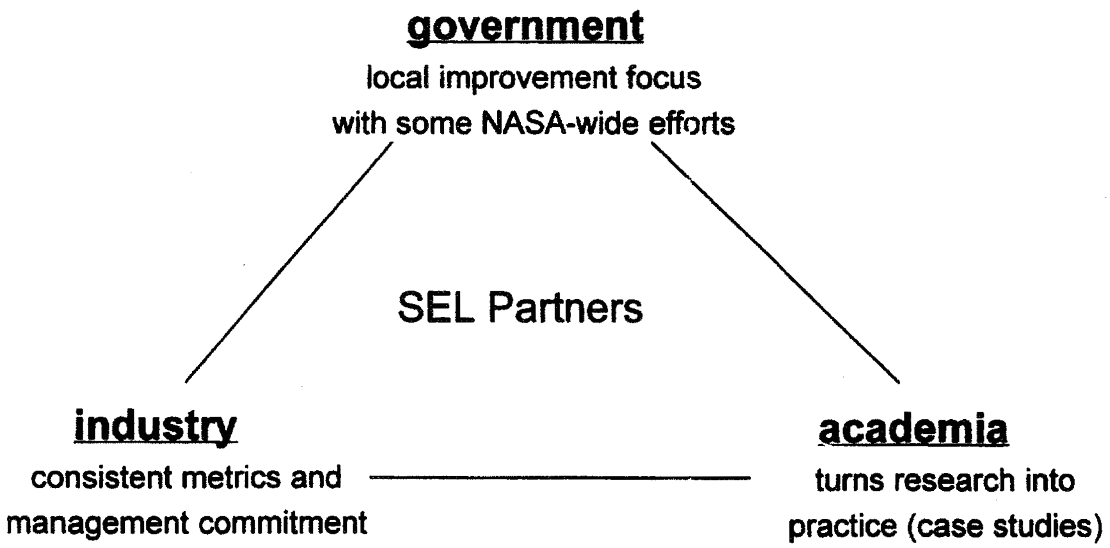
NASA & CSC

SEL Data Base

Asset Library

- **Measures**
- **Reuse info**
- **S/w eng reports**
- **Project records**

Work Planning Approach - Current



government
local improvement focus
with some NASA-wide efforts

SEL Partners

industry
consistent metrics and
management commitment

academia
turns research into
practice (case studies)

- **Board of Directors: 4-6 senior leads**
- **Annual proposals: based on local improvement goals**

Work Planning Approach - Future Concept

Team	Remain self-directed team
Partnership	Broaden to include other industry and academic partners
Board of Directors	Enterprise representation from projects Proposals linked to organizational & sponsor goals

SEL Infrastructure - Current

- Experience repository
  - Increased measurement database capabilities (e.g., COTS process & product data)
  - Online/Web-based data collection tools
- Project Support
  - Earlier team participation (requirements generation)
  - Feedback reporting

SEL Infrastructure - Future Concept

Database	Collect & analyze: team-specific activity data acquisition-related data
Project Support	Model effort/ schedule/ errors for: development, integration & maintenance Online report distribution

Experimental Adaptations - In Progress

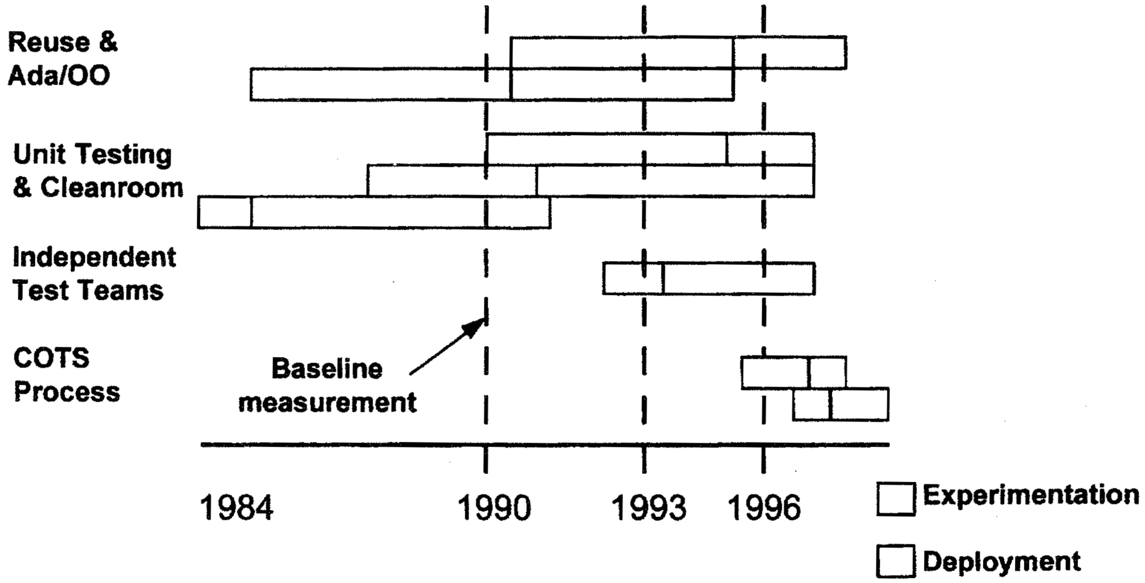
- Study approach and selection
  - Smaller, overlapping studies
  - Team & individual processes considered
  - Firmer link between experimental goals & measures
- Analysis of results
  - Faster model building
  - Feedback to current teams
  - Frequent baselines for managers

Accelerating Analysis

- Use of qualitative techniques
  - Focused interviews
  - Quick-look analysis
    - Interviews
    - High-level profile data
- Interim analysis reports
  - Online feedback replacing meetings
  - Reports build on previous

Overlapping Studies

Length of studies shortens - Baseline period decreasing



Technology Transfer Approach (through 1996)

- Depended on answers to a few simple questions
  - Scope?
    - Internal - more "hands on"
    - External - written guidance
  - Sponsor?
- Transferred products
  - Guidebooks, training
  - Tools

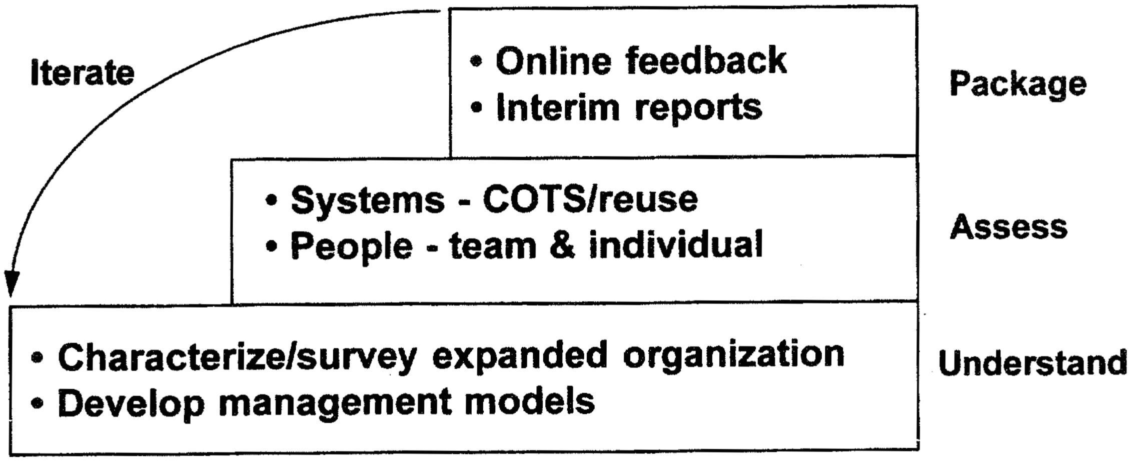
Technology Transfer Approach Evolving

- Domain-based "filter"
  - Consider a range of factors
    - Organization, environment
    - More questions at the start
  - Predict success of transfer
  - Pilot in place at JSC
- Integrate with other NASA/industry efforts
  - Filter best practices quickly
  - Use existing process/product improvement groups

Next Steps for the SEL

1. Baseline/profile expanded organization
   - Processes used
   - Products generated
   - Report to organization
2. Select projects for focused support
   - Development/integration/maintenance
   - Process technologies (ongoing work)
3. Evolve SEL direction
   - Enterprise representation
   - Investigate other partnerships

The SEL Within Project Goddard



- Online feedback
- Interim reports

- Systems - COTS/reuse
- People - team & individual

- Characterize/survey expanded organization
- Develop management models

and the 'beat' goes on ...

The Package-Based Development Process in the Flight Dynamics Division

Amalia Parra, Computer Sciences Corporation
Carolyn Seaman, University of Maryland
Victor Basili, University of Maryland
Stephen Kraft, NASA/Goddard Space Flight Center
Steven Condon, Computer Sciences Corporation
Steven Burke, Computer Sciences Corporation
Daniil Yakimovich, University of Maryland

Abstract

The SEL has been operating for more than two decades in the FDD and has adapted to the constant movement of the software development environment. The SEL's Improvement Paradigm shows that process improvement is an iterative process. *Understanding, Assessing* and *Packaging* are the three steps that are followed in this cyclical paradigm. As the improvement process cycles back to the first step, after having *packaged* some experience, the level of *understanding* will be greater. In the past, products resulting from the *packaging* step have been large process documents, guidebooks, and training programs. As the technical world moves toward more modularized software, we have made a move toward more modularized software development process documentation, as such the products of the *packaging* step are becoming smaller and more frequent. In this manner, the QIP takes on a more spiral approach rather than a waterfall.

This paper describes the state of the FDD in the area of software development processes, as revealed through the *understanding* and *assessing* activities conducted by the COTS study team. The insights presented include: (1) a characterization of a typical FDD COTS intensive software development life-cycle process, (2) lessons learned through the COTS study interviews, and (3) a description of changes in the SEL due to the changing and accelerating nature of software development in the FDD.

1 Background

The Flight Dynamics Division at NASA/Goddard Space Flight Center has had a history of effective reuse of software to levels as high as 90%. The increase has been affected by the use of Ada and object-oriented technologies. This experience led to the creation and use of an architectured component library for a certain class of systems so that these systems could be "configured" rather than developed [Condon et al., 1996]. It has also motivated the outsourcing of software development for more "standard" systems, which in turn has led to a move from internal reuse to the use of external software packages. The introduction of package-based software development—rapid configuration of software systems based on Commercial-Off-The-Shelf (COTS) packages, Government-Off-The-Shelf (GOTS) packages, and some custom-built reusable packages—has motivated the Software Engineering Laboratory (SEL) to provide guidance in this new era by updating the *SEL Recommended Approach to Software Development* [SEL, 1992]. Before updating this important SEL guidebook, however, it was first necessary to understand and improve this new package-based process within the flight dynamics domain.

The traditional SEL approach to software improvement involves three steps. These are described in more detail in Section 2, but briefly they are as follows: (1) *understand* the current situation in the local environment (for us, the FDD) and develop appropriate goals for improving specific items; (2) *assess* how to achieve these goals by defining process changes, testing them on one or more projects, and analyzing the results of this experiment; (3) package the lessons learned from step 2 and integrate these into the local software development process. In any given SEL experiment, this 3-step improvement process is generally a cyclic one, involving several iterations. In addition the steps can overlap somewhat.

The first phase of the SEL COTS study was conducted during the last few months of 1995. Because the FDD had limited experience developing COTS-based systems at that time, the SEL looked at experiences of outside organizations in order to understand the challenges associated with this type of development and to gather best practices used on COTS-based projects. Using a solid understanding of the FDD project domain, history and environment, the SEL synthesized this information into a strawman process to be used to produce COTS-based systems in the FDD. This initial strawman process was then reviewed for feasibility by key FDD software engineers (both civil servant and contractor) who have had some experience with COTS. The resulting strawman process, presented in the *Packaged-Based System Development Process* [Waligora, 1996], is available on the SEL's Web page, http://fdd.gsfc.nasa.gov/selres.html.

As more FDD projects began using COTS to construct their software systems, the next phase of the SEL COTS study began. The goals of this phase, which is the subject of the rest of this paper, include gathering a current understanding of COTS-based project, suggesting areas of improvement for further study, and providing guidance to current and future COTS-based projects.

In section 2, we present some of the terminology used in the rest of the paper. Section 3 describes the approach we used in the study, and section 4 describes the COTS-based software development process that emerged from the data we collected, as well as some insights into that process. Section 5 describes some of the steps that the SEL has taken to keep up with the pace of change, in particular in packaging the results of the COTS study in a timely and relevant manner. Section 6 describes some of the future plans for this line of investigation.

2 Terminology

The words "Commercial-Off-The-Shelf" are very generic; they can be used to in reference to many different types and levels of software, e.g. software that fills a specific functionality or a tool used to generate code. In this paper the term *COTS* implies a COTS product that has specific functionality as part of a system — not merely a tool, but a piece of 'pre-built' software that is *integrated* into the system and must be *delivered* with the system to provide operational functionality or sustain maintenance efforts.

The term *COTS project* refers to a project that integrates COTS packages and other software to develop a system. This is not to be confused with the development of COTS packages that occurs at the 'vendor' corporation.

Additionally, the term GOTS is equivalent to COTS in this study because the process followed is for the most part identical to developing a system with COTS.

3 Experimental Approach

The Improvement Paradigm, shown in Figure 1, is a SEL tool for process improvement and is commonly used to plan SEL studies. The SEL COTS study team used this concept to guide its work. The paradigm is a three step, iterative process. The basic step is understanding, which identifies the current status of some aspect of software development in the SEL. The next step is assessing, which determines potential improvements. The main activities of the COTS study team are primarily focused on this assessing step. Packaging is the top step, in which improvements are documented and integrated into the environment to form the basis for the next level of understanding.

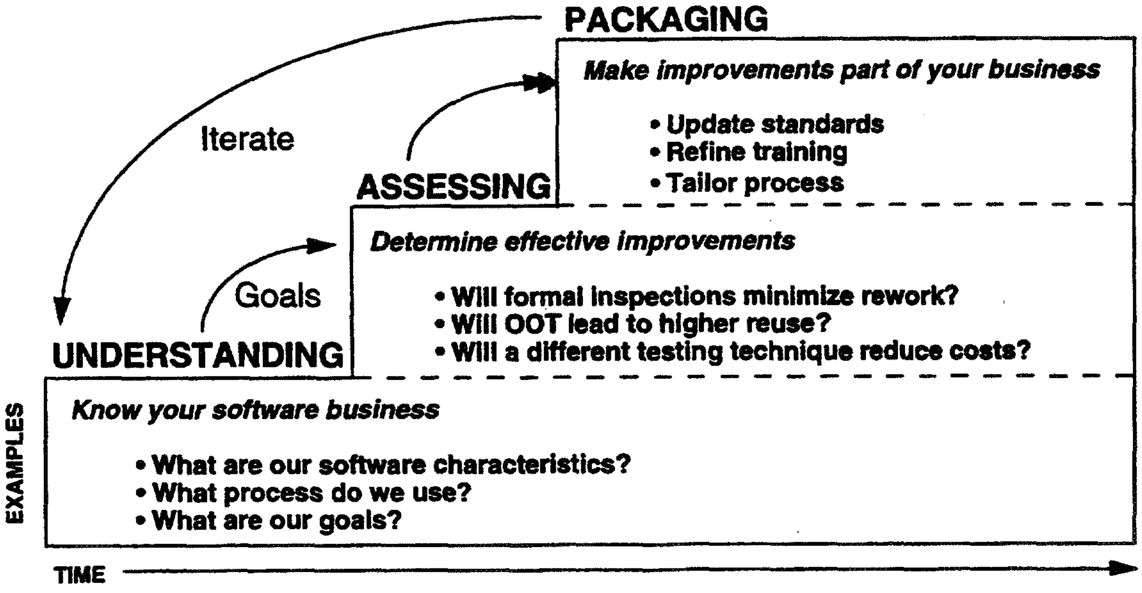

PACKAGING
Make improvements part of your business
• Update standards
• Refine training
• Tailor process
Iterate
ASSESSING
Determine effective improvements
• Will formal inspections minimize rework?
• Will OOT lead to higher reuse?
• Will a different testing technique reduce costs?
Goals
UNDERSTANDING
Know your software business
• What are our software characteristics?
• What process do we use?
• What are our goals?
EXAMPLES
TIME

Figure 1. The Improvement Paradigm

The initial understanding step for the COTS study was a series of interviews with representatives from 12 COTS projects. Based on the interview data, we then described the COTS-based development in the FDD, presented in section 4.1. Also based on interview data, we redesigned the way development effort data is collected in the FDD. These two activities made up our assessing step. These results were packaged, in the packaging step, through updated data collection forms (described in section 5.2) and study briefs (described in section 5.3). The next level of understanding, then, is provided by the baseline process description and data from the new effort forms.

4 COTS-based Software Development Process

As a first step in understanding where COTS-based development in the FDD stood, the study team analyzed the current data collection. Historically the SEL collects effort data. For typical pre-COTS era projects the SEL has a baseline of effort divided into four simple categories of activities. The SEL anticipated need for data specific to COTS projects, made an attempt to gather data on this effort, but the level of detail was too general to allow understanding of the COTS-related effort. One indication that the SEL was not capturing useful data is the large amount of effort that fell into the "other" category.

Clearly, the quantitative information available was not sufficient for us to identify and understand the new issues that were arising in relation to the use of COTS packages in FDD projects. In order to gather more and richer information on this topic, the study team designed and conducted structured interviews, using three levels of interview guides at increasing levels of detail, with representatives from 12 projects. Topics covered included the process steps carried out, what problems were encountered with the use of COTS in development, and how the incorporation of COTS has changed the software development process.

4.1 Process Description

Our interviews uncovered the new process flow, shown in Figure 2. The study team discovered more complexity in the current practice than expected in theory. For example, we had expected vendor interaction to be simple, and to end with the purchase of a product. In reality, the interaction continues throughout the life cycle and the flow of information is not merely one way. Surprisingly, we found a strong dependence on *bi-directional* information flow. Also shown is a more constant involvement with separate organizations, such as other projects that also use COTS, independent evaluation teams, and other customers of the vendor. Portions of the COTS-based systems include traditional developed software. So an issue to consider is how to fit together our traditional process, as documented in the SEL Recommend Approach to Software Development, and our new way of doing business by integrating COTS packages to build a system.

The software development teams interviewed included both FDD and CSC personnel. Although not every team followed all of the steps outlined below, a composite process flow emerged from the interview data. Note: None of the project teams interviewed had begun sustaining engineering. This step will be evaluated in future studies. The steps in the overall process, as shown in Figure 2, are as follows:

- Requirements Analysis
- Package Identification, Evaluation and Selection
- Non-COTS Development
- Glueware Requirements and Development
- System Integration and Test
- Target System Installation and Acceptance Test
- Discrepancy Resolution
- Sustaining Engineering

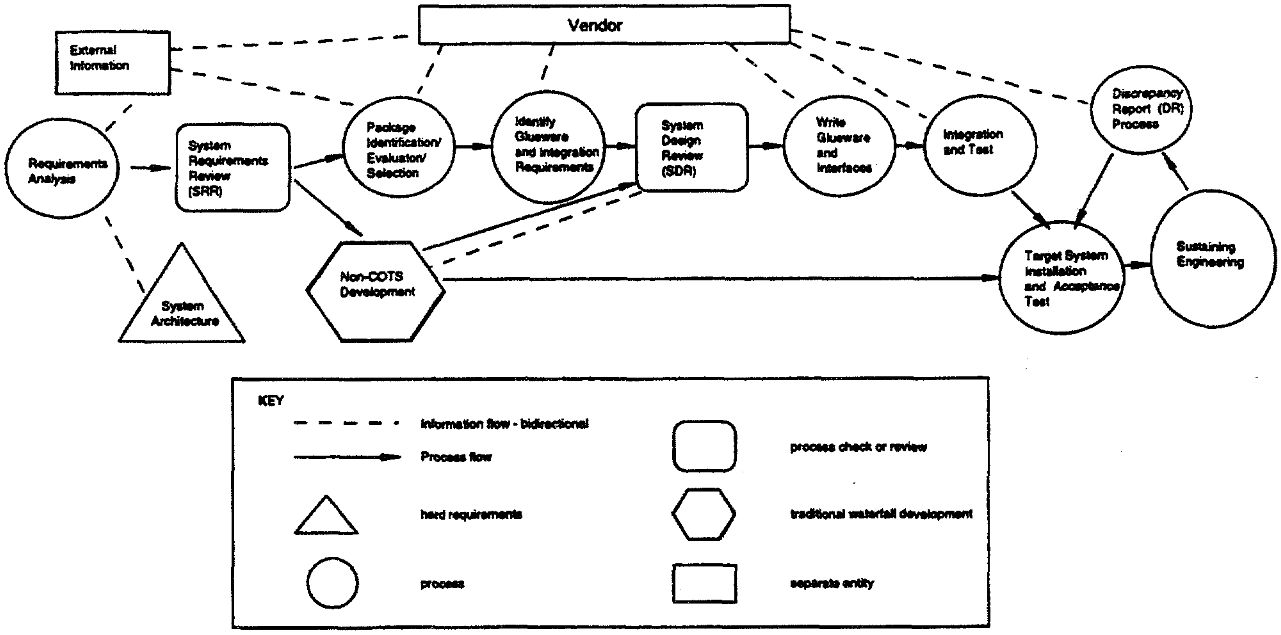

Vendor
External Information
Requirements Analysis
System Requirements Review (SRR)
Package Identification/ Evaluaton/ Selection
Identify Glueware and Integration Requirements
System Design Review (SDR)
Write Glueware and Interfaces
Integration and Test
Discrepancy Report (DR) Process
System Architecture
Non-COTS Development
Target System Installation and Acceptance Test
Sustaining Engineering
KEY
information flow - bidirectional
Process flow
hard requirements
process
process check or review
traditional waterfall development
separate entity

Figure 2. Process Flow for COTS Projects

The earliest steps in COTS-based development are similar to traditional development - requirements gathering. In the requirements phase a strong emphasis is on gathering external information. Much of this information comes from separate organizations, particularly the product vendor, in the form of documented functionalities. Some project requirements are predefined, with minimal requirements analysis needed. Early reviews of the requirements are crucial even with a less formal process.

Following requirements analysis are the new and concurrent steps of package identification, evaluation, and selection. These are new activities, requiring new technical skills and new administrative duties, especially in the area of procurement.

Package identification consists of Web searches, product literature surveys and reviews, other system component reuse, and recommendations from external sources. Product information is kept in a central justification notebook, or an evaluation notebook. Not only are product evaluation notes kept, but subjective comments concerning the vendor quality and responsiveness are kept, too.

As packages are identified, the evaluation and selection processes begin. Package evaluation steps mentioned in the interviews consisted of prototyping, vendor demonstrations, and in-depth review of literature such as manuals and user guides. Glueware and interfaces as dictated by the system architecture, operating system and hardware are identified. Vendor training, sites, and availability, are considered. Procurement issues surface such as development fees for added requirements, licensing and maintenance fees and sustaining engineering support.

The selection step sometimes uses a weighted average. To do this, vendor capabilities are listed and mapped to the system requirements. With team agreement, weights of importance are assigned to each requirement. Then each team member votes. Team members are polled and the votes tallied. Discussion ensues and a choice is made. In cases where the vendor will code additional functionality, the vendor is notified of the decision. In the case of one team, when the vendor was told they were selected, the vendor announced a hidden cost. Negotiations ended altogether, and the second choice vendor and package were used.

In both of these first two process stages, we found that some projects relied on the COTS Evaluation Team, which is chartered by the parent organization to survey the marketplace and evaluate vendor packages that fall within the domain expertise of the mission team's organization. The evaluation team then reports its findings and offers this knowledge to the project teams. The project team is ultimately responsible for deciding what package to select and integrate. The evaluation team can be important when delivery time is driving the project - time the development team doesn't have for product evaluations.

Most projects studied have an element of traditional development that does not depend on COTS or other packages. This development begins in parallel with the early COTS-related steps, as a traditional development project. Non-COTS cost and schedule are monitored. There is a bi-directional information flow between the COTS-based process flow and the non-COTS development that comes into play in the design review. Only some teams held a formal System Design Review (SDR), but all teams mentioned some mechanism to apprise the customer of the design.

After the design review, whether it is formal or informa., traditional non-COTS development continues in parallel with the coding of the glueware and the interfaces. Close contact with the vendor technical staff, or a competent Help Desk is essential during this development.

The integration step varies a great deal from project to project, depending on which and how many COTS products are being used. At system integration and testing the COTS packages are treated as black-boxes. The teams commented that testing focused on the interface glueware and the input file format. Again, the importance of the vendor technical staff or Help Desk availability was emphasized. Testing is conducted on each software component as the components are integrated, piece-by-piece.

Unlike the traditional life-cycle, no formal acceptance testing or operational readiness reviews were mentioned by the teams. The development team installs the software on the target system. Once installed, navigational training to familiarize the customer with the system is conducted. During this phase, a member of the development team is the single point-of-contact or intermediary between the customer and the vendor. This person is responsible for reporting discrepancies, and handling software "patches" or corrections. Interviewees mentioned that software patches were placed on vendor Web sites that were downloaded to the target system.

The end of the configuration process is marked by the sustaining engineering effort. To date no team that the study team interviewed had reached the sustaining engineering stage.

4.2 Lessons Learned & Experience Gained

The developers interviewed were also asked to describe the major differences between COTS-based development and traditional development, and the advantages and drawbacks. Some mentioned the obvious difference, i.e. that there is now a whole lot of software that doesn't need to be implemented. It's no longer the task of building a big system, but of using already-built pieces. But there were other less obvious differences. Some of those differences mentioned were:

- different design phases
- looser process requirements
- new or greatly increased need for vendor interaction
- procurement skills now needed
- new or greatly increased need for product evaluations
- no unit test or inspections of packaged software

Advantages of COTS-based development that were mentioned included*:

- more flexible requirements
- less process overhead
- less code to write
- less debugging
- shorter cycle time
- better adherence to schedule.
- serendipitously useful functionality in COTS packages

* Note: "Shorter cycle time" and "less process overhead" may be due to the pressure to do things faster as much as due to the adoption of COTS.

Many of the disadvantages mentioned had to do with dealing with the vendor, including the risks of less than full knowledge beforehand, dependence on the vendor, and vendor negotiations. Another disadvantage, which some people listed as an advantage, is the relative looseness of the process in package-based projects. Some people thought that more rigor was needed.

5 Packaging the Approach

FDD projects have moved rapidly from a reuse-based development process to a COTS-based system development process. The SEL needed to react quickly with new mechanisms to adapt to these and other changes in the environment. The changes that the SEL has undergone are important to study because the nature of software development is changing and will require further changes in the research methods of the SEL and other organizations. As we encounter new problems, we need new ways to address these issues.

We will address the natural adaptation of the SEL that is taking place; the learning, refitting and adjusting of the SEL learning procedures in order to keep pace with the new organization and environment. Three major innovations in standard procedures will be discussed :

1) The use of qualitative analysis, mostly in the form of structured interviews and analysis of data gathered from those interviews.
2) Changes to the database in terms of what is being collected and analyzed in order to keep track of the changing business in FDD.
3) The generation of *Study Briefs*, which are short, quickly disseminated communications on a variety of topics—lessons learned, early analysis results, definitions of new terms, etc.—to keep information flowing between the EF and the project organization in a timely manner.

The use of qualitative analysis was necessitated by the COTS study. The study team found that with this change in technology, the quantitative data that the SEL collects does not tell the entire story of what is occurring on projects. During the course of these interviews the SEL team members interacted with the technical personnel. These interactions led the SEL to realize the need for more effective, frequent communication: (1) communication from the SEL to the project organization about what the SEL was learning, and (2) feedback from the project organization to the SEL to corroborate and refine the SEL's evolving models. This realization became the catalyst for the SEL *Study Briefs*. Interviews specific to the COTS study also showed that the data collected for COTS was insufficient. This sparked the modification to the Weekly Effort Form to include COTS specific details. The transition to this new form has been simple due to the new re-engineered SEL database, that has been revolutionized using COTS products and transitioned to a workstation platform. This allows us to use the database as a repository for information on the COTS products used as well as the effort involved in putting together a COTS based system.

5.1 Interviews and Qualitative Data

Empirical studies in software engineering, like the ones that the SEL has engaged in for two decades, have traditionally relied on standard quantitative methods in order to characterize some aspect of a software development process. In some cases, several quantitative studies of various sizes and scopes have been conducted to address one general issue, e.g. Cleanroom software development [Selby et al., 1987]. Approaching a problem from several angles in this way yields a more complete description of a particular process or of the effect of a particular technology. This approach has helped the SEL and other organizations learn a great deal about their software business. In recent years, however, software projects in the SEL environment have become both more complex and faster-paced, as is true in much of the software industry. This has motivated the SEL to find ways to provide richer answers to more complex problems in less time.

One approach to achieving this is to use different research methods than the SEL is accustomed to using, in particular qualitative methods. Qualitative data is information in the form of words and pictures, as opposed to quantitative data, which is in the form of numbers. Qualitative analysis is simply the examination and analysis of qualitative data in order to form conclusions and hypotheses. Qualitative data is by definition richer and carries more information than quantitative data. On the other hand, it is more complex and harder to analyze. Qualitative analysis methods have been designed to deal with this complexity [Glaser and Strauss, 1967]. Combinations of qualitative and quantitative methods are especially useful because the two types of methods tend to deal with the complexity of the subject in complementary ways.

The COTS study is one of the first SEL studies to use qualitative data to a large extent. The qualitative data used in this study comes from extensive interviews with software developers and managers. Using this data has allowed an in-depth examination of COTS-based development that incorporates a variety of perspectives in one study. For example, data was collected on the problems encountered during COTS-based development, the different steps involved, the parts of the process which are effort-intensive, and the roles that must be filled to carry out this type of development. Much of this information would be very difficult to collect quantitatively, and would have required multiple studies, each measuring various attributes in different ways.

The drawbacks to doing qualitative study is that it doesn't provide "hard" results in terms of easy-to-use mathematical models (e.g. regression models) or easy-to-summarize relationships between variables (e.g. correlations). Instead, qualitative results are more complex, "messier", to reflect the complexity of the problem being described.

Qualitative data, mostly from interviews, is also being used to some extent on other ongoing SEL studies. In combination with other quantitative methods, we believe the use of qualitative analysis in current and future studies will help the SEL provide the development community with more useful, in-depth, and realistic explanations of software development phenomena.

5.2 New Data Forms - Quantitative Data

In response to a need for more COTS related data, the SEL realized an opportunity to update the types of data that are maintained in the SEL database. This was accomplished by the modification of an existing form, the Weekly Effort Form, and the addition of a new form, the COTS & Tools Information Form.

5.2.1 Weekly Effort Form

As the interview data was leading us to define the COTS-based development process, the study team saw that there were new activities that projects were conducting. These include:

COTS/GOTS Evaluation
COTS/GOTS evaluation activities included identifying packages, collecting information, attending demos, evaluating and selecting COTS/GOTS packages.

COTS/GOTS Integration
This included integrating COTS/GOTS, possibly with other software components, to produce individual applications or subsystems. This also included the writing and debugging of glueware.

COTS Package Familiarization
Package familiarization is spending time to learn to use a COTS package, not including formal training, which would be included under other effort categories, nor package familiarization for the purposes of evaluation.

Configuration Management
Configuration management had not previously been a separate category.

Procurement
This included procuring and purchasing packages, interacting with the vendor regarding licensing and maintenance agreements, etc.

These new activities were merged into the Weekly Effort Form (WEF), the existing SEL form for collecting effort data from the technical personnel. This merger created a WEF modified for COTS that was then used on a trial basis by two projects. (See Appendix A for the original WEF and Appendix B for the experimental COTS WEF.) After experimental use of this COTS WEF, and a few resulting updates, the SEL decided to implement the updated WEF across the organization. This was accomplished through full consultation with FDD technical personnel. The resulting WEF was put into place November 1997 across the organization (see Appendix C).

The graph shown in Figure 3 indicates the type of data collected by the experimental COTS WEF, the WEF which was introduced in October 1995 (and which had only a single COTS activity category), and the even earlier SEL weekly effort form which was in use prior to October 1995. The leftmost bar shows the typical distribution of effort on completed FDD projects prior to October 1995. The major activities are *design*, *code*, *test*, and *administrative*; none deal with COTS.

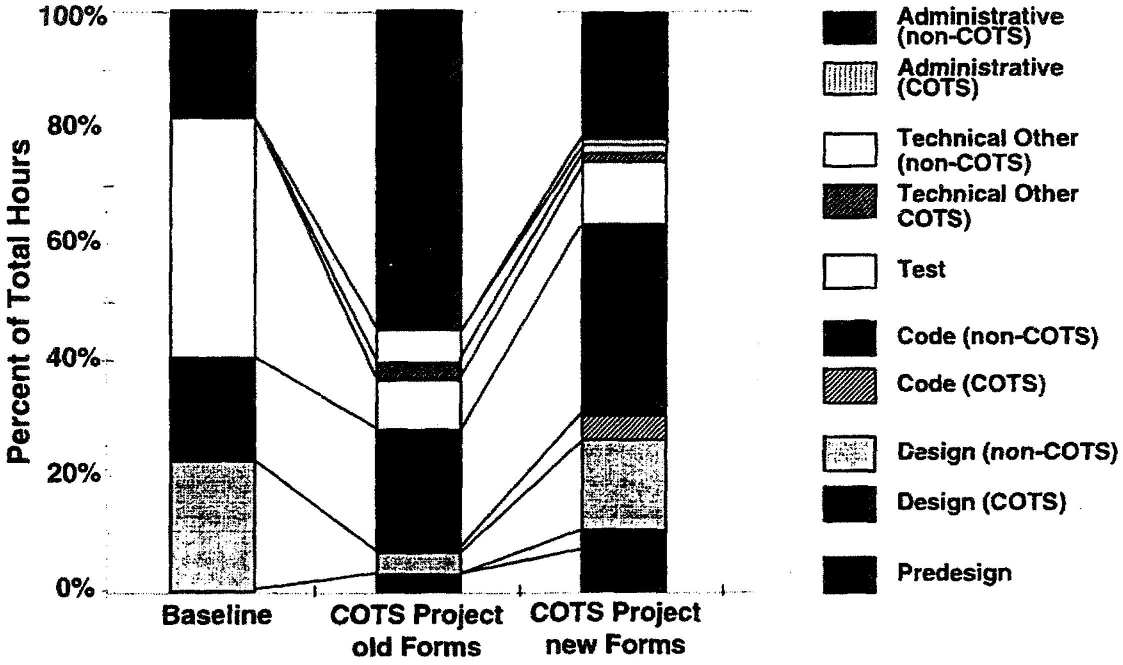


Figure 3. COTS Data From Projects

The middle bar shows the effort distribution for a nearly complete FDD project that was developing during the era of the WEF that was introduced in October 1995 and which involved some COTS integration. This WEF introduced a *predesign* category. It also introduced a single *COTS* activity and a so-called *technical other* category. Note that this bar shows a great increase in the proportion of project effort spent in the *administrative* activity. Various hypotheses were examined to explain this change, but none proved conclusive.

The rightmost bar shows the distribution of effort for a FDD project that involved a fair amount of COTS integration but which was only partially complete. This project began using the experimental COTS WEF soon after the project began. Only about twelve weeks of this project's effort data were available for analysis for this paper. The data in this bar is thus insufficient to draw any conclusions on the distribution of effort on a typical FDD project, yet alone a project in another environment. Data on several complete projects would be required before the typical FDD effort distribution on a COTS project could be determined.

Future studies are underway to determine this and to address more specific issues. These are described in Section 6.

5.2.2 COTS & Tools Information Form

In order to collect context data about the COTS packages used on projects, the SEL developed the COTS & Tools Information Form (CTIF), shown in Appendix D. The need for the CTIF became evident during the interview process. We were collecting qualitative data, such as which COTS packages are used, what support is provided by the vendor, and whether it is embedded into the system or merely a tool. Rather than maintaining all this information in the interview notes, we developed the CTIF to collect data that would be stored, and readily accessible, in the SEL database. Using the CTIF to collect this context data allows us to characterize the COTS products in order to better compare projects that are related either in the type of COTS products used, or in functionality provided by COTS.

5.3 Study Briefs

The SEL realized a need for compact products. *SEL Study Briefs* are an example of this as they concisely document and distribute information that might fall through the cracks. A Study Brief is less than a process document, yet much more than informal communications. The modularity of the Study Briefs allows the user community to incorporate "one page worth of process" into their busy schedules. Study Briefs also serve as a tool for communication with the inclusion of the technical community in the feedback loop section. A sample Study Brief is shown in Appendix E.

The format of a SEL Study Brief is shown Table 1.

Study Brief Field	Explanation of Field
Study Brief Number	number assigned (in order of posting on web)
Issue	topic of Study Brief
Purpose	goal of Study Brief
Current Understanding	body of the Study Brief, may vary in style
Feedback	comments received from audience after Study Brief has been posted to web site
Original Author	person or persons originating Study Brief
Responsible Author	person responsible for receiving feedback, and possibly modifying the Study Brief (in most cases same as Original Author)
Contributors	others who contributed to the Study Brief
References/Relevant Links	materials used in preparation of Study Brief, additional information on a topic, or hot links to on-line sources
History	first published date, any revision dates

Table 1. SEL Study Brief Format

6 Future Directions

After analysis of the current process and review of the issues that are most relevant to this new COTS-based development environment, several topics for further study have been identified:

1) The long-range effects of COTS use, in particular the maintenance of systems which incorporate COTS packages,
2) Modeling and estimating effort, cost, and schedule of COTS projects based on data collected with the new forms,
3) Risks of COTS use, to be studied with a series of SEL case studies, possibly including the re-engineered SEL database development, and
4) Methods for measuring the "distance" between a set of requirements for a new system and the available COTS packages which could be used to satisfy those requirements.

The selection and implementation of COTS is easier to understand than is the maintaintenance of a COTS-based system. Such a system will require modifications and enhancements during its lifetime, and many of these modifications may be prompted by vendor updates to the COTS packages. Maintenance includes less obvious costs; maintenance agreements and licensing are more tangible than the effort that will be required to identify the parts of the code that are affected by a small change elsewhere so that the modification to one area does not cripple the system. The projects interviewed for the COTS study had not moved into a maintenance phase. The SEL sees this as an area for further research.

The SEL has a long-standing tradition in the FDD for providing models for the estimation of effort, cost and schedule. The interviews uncovered a need for new models to support COTS projects. The SEL has begun efforts to baseline the current situation across the organization. The next steps toward developing models include collecting a reasonable amount of data from which to draw quantifiable conclusions.

The interviews identified risk as an important topic. The SEL determined that gathering case studies of various COTS-based systems with emphasis on the risks expected, as well as the risks involved, would provide valuable information. It appears likely that many of the risks of introducing COTS systems are domain-independent. Because of this the SEL's recently re-engineered software metrics collection and reporting system would be a good non-FDD system to examine as a case study of COTS risks. In this re-engineering process, the SEL upgraded the SEL's COTS relational database management system and added an additional COTS product to automate the submission of data to the SEL by users.

In related research, we seek to develop a mechanism to measure the "distance" between the need for functionalities (the requirements) and the specification of available COTS. Such "functional distance" measures should help us to predict the amount of glueware necessary to integrate COTS with the rest of the system. The costs of glueware is one key factor in the total cost of using COTS software.

Appendix A - Old WEF (Introduced October 1995)

WEEKLY EFFORT FORM

Use this form to record all hours you worked during the week.

Name:

Project:

Date (Friday):

For Librarian's Use Only

Number: ____________

Date: ____________

Entered by: ____________

Checked by: ____________

Application or Subsystem (or "N/A" as appropriate.)						
Release/Build Number (or "N/A" as appropriate.)						
SCR Number (or "N/A" as appropriate)						
Hours By Activity (List hours in a separate column for each application, release/build and SCR combination.)						
PRE DESIGN	Requirements Spec. Definition/ Development	Hours spent defining and developing the requirements specifications				
	Requirements Analysis	Hours spent understanding requirements specs or understanding SCRs for enhancements or adaptations				
	Error Analysis/ Debugging	Hours spent finding a known error in the system; may be in response to SFR, STR, SCR. (includes generation and execution of tests associated with finding the error)				
	Impact Analysis/Cost Benefit Analysis	Hours spent analyzing several alternative implementations and/or comparing their impact on schedule, cost, and ease of operation				
DESIGN	Design Creation or Modification	Hours spent developing or changing the system, subsystem, or component design (includes development of PDL, design diagrams, meeting materials, etc.)				
	Design Review/ Inspection	Hours spent reading or reviewing design (includes design meetings and consultations, as well as formal and informal reviews, walkthroughs, and inspections)				
CODE	Code Generation/ Modification	Hours spent actually coding system components (includes both desk and terminal code development)				
	Code Review/ Inspection	Hours spent reading code (for any purpose other than isolation of errors) or inspecting other people's code				
	Unit Testing	Hours spent testing individual components of the system (includes writing test drivers and informal test plans)				
TEST	System Integration/ Integration Testing	Hours spent integrating components into the system; hours spent writing and executing tests that integrate system components (includes system tests)				
	Regression Testing	Hours spent regression testing the modified system				
	Independent Testing Support	Hours spent supporting independent testing, including training of testers				
MISC	Prototyping	Hours spent prototyping to investigate a particular issue (not to be confused with other activity hours when the entire system is a prototype)				
	COTS/GOTS	Hours spent evaluating, selecting, procuring, integrating and testing COTS/GOTS products				
OTHER	Documentation	Hours spent creating and reviewing deliverable documents				
	Training for Self	Hours spent taking courses (including computer-based training), attending seminars, etc.				
	User Support/Training	Hours spent training users and responding to their questions				
	Management	Hours spent managing or coordinating work and reporting status				
	Other	Other development hours not covered above				
▨	Total	Total hours per column	0.0	0.0	0.0	0.0
▨	Grand Total	Total hours	0.0			

Appendix B - Experimental COTS WEF

For Librarian's Use Only
Number:
Date:
Entered by:
Checked by:

Experimental "COTS modified" WEEKLY EFFORT FORM

Use this form to record all hours you worked during the week.

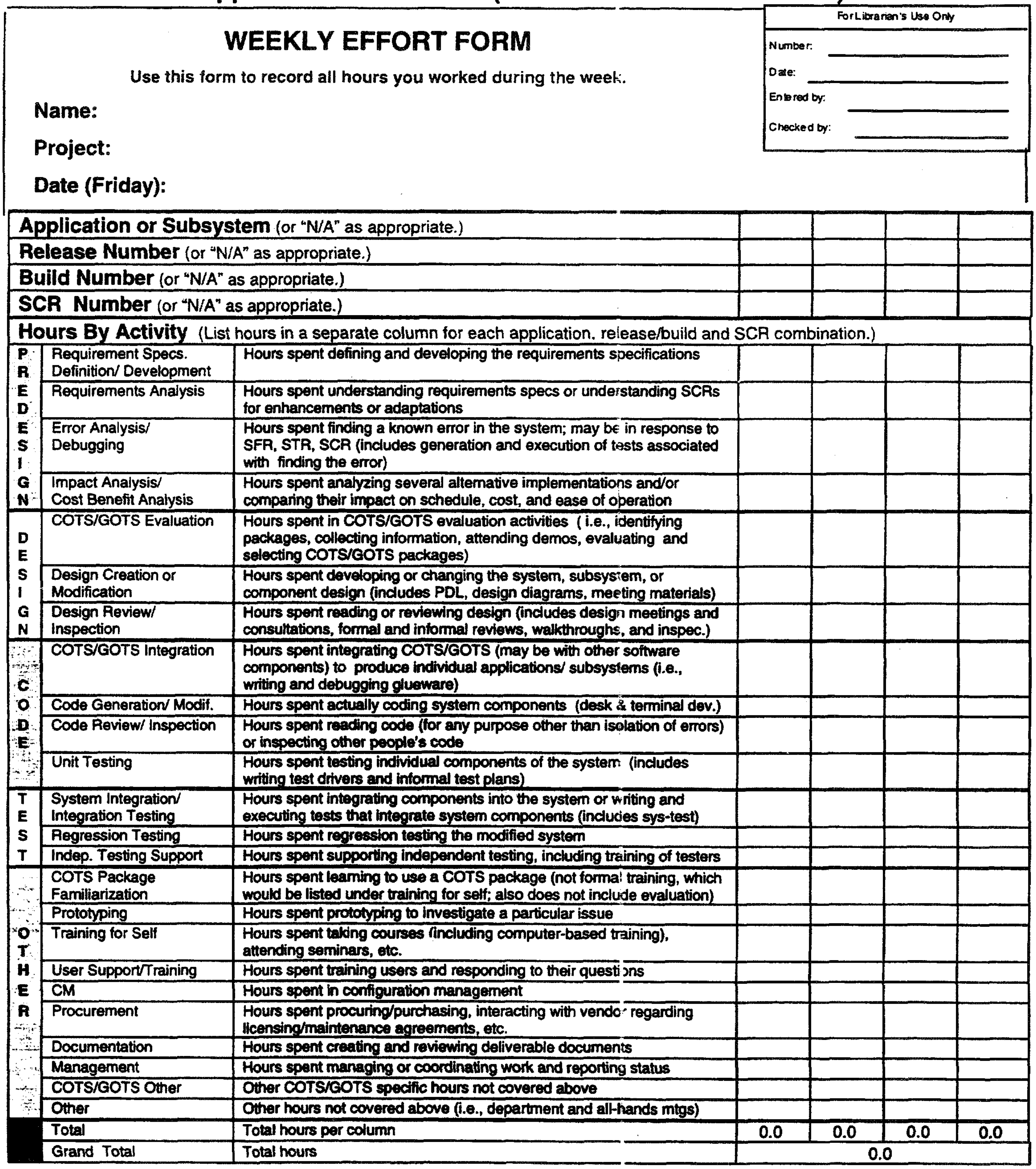
Name:

Project:

Date (Friday):

Application or Subsystem (or "N/A" as appropriate.)						
Release Number (or "N/A" as appropriate.)						
Build Number (or "N/A" as appropriate.)						
SCR Number (or "N/A" as appropriate)						
Hours By Activity (List hours in a separate column for each application, release/build and SCR combination.)						
PREDESIGN	Requirement Specs. Definition/Development	Hours spent defining and developing the requirements specifications				
	Requirements Analysis	Hours spent understanding requirements specs or understanding SCRs for enhancements or adaptations				
	Error Analysis/ Debugging	Hours spent finding a known error in the system; may be in response to SFR, STR, SCR.				
	Impact Analysis/Cost Benefit Analysis	Hours spent analyzing several alternative implementations and/or comparing their impact on schedule, cost, and ease of operation				
DESIGN	COTS/GOTS Evaluation	Hours spent in COTS/GOTS evaluation activities, (i.e., identifying packages, collecting information, attending demos, evaluating & selecting COTS/GOTS packages)				
	Design Creation or Modification	Hours spent developing or changing the system, subsystem, or component design (includes development of PDL, design diagrams, meeting materials, etc.)				
	Design Review/ Inspection	Hours spent reading or reviewing design (includes design meetings and consultations, formal and informal reviews, walkthroughs, and inspections)				
CODE	COTS/GOTS Integration	Hours spent integrating COTS/GOTS (& other software components) to produce individual applications/ subsystems (i.e. writing & debugging glueware, COTS package familiarization)				
	Code Generation/ Modification	Hours spent actually coding system components (includes both desk and terminal code development)				
	Code Review/ Inspection	Hours spent reading code (for any purpose other than isolation of errors) or inspecting other people's code				
	Unit Testing	Hours spent testing individual components of the system (includes writing test drivers and informal test plans)				
TEST	System Integration/ Integration Testing	Hours spent integrating components into the system; hours spent writing and executing tests that integrate system components (includes systest)				
	Regression Testing	Hours spent regression testing the modified system				
	Indep. Testing Support	Hours spent supporting independent testing, including training of testers				
MISC	Procurement	Hours spent procuring/purchasing, interacting with vendor regarding licensing/maintenance agreements etc.				
	Prototyping	Hours spent Prototyping to investigate a particular issue				
OTHER	Documentation	Hours spent creating & reviewing deliverable documents				
	Training for Self	Hours spent taking courses (including computer-based training), attending seminars, etc.				
	User Support/Training	Hours spent training users and responding to their questions				
	Configuration Mgmt.	Hours spent in configuration management				
	Management	Hours spent managing or coordinating work and reporting status				
	COTS/GOTS Other	Other COTS/GOTS specific hours not covered above				
	Other	Other development hours not covered above				
▨	Total	Total hours per column	0.0	0.0	0.0	0.0
▨	Grand Total	Total hours	0.0			

Appendix C - New WEF (Introduced November 1997)



For Librarian's Use Only

Number: __________

Date: __________

Entered by: __________

Checked by: __________

WEEKLY EFFORT FORM

Use this form to record all hours you worked during the week.

Name:

Project:

Date (Friday):

Application or Subsystem (or "N/A" as appropriate.)						
Release Number (or "N/A" as appropriate.)						
Build Number (or "N/A" as appropriate.)						
SCR Number (or "N/A" as appropriate.)						
Hours By Activity (List hours in a separate column for each application, release/build and SCR combination.)						
PREDESIGN	Requirement Specs. Definition/ Development	Hours spent defining and developing the requirements specifications				
	Requirements Analysis	Hours spent understanding requirements specs or understanding SCRs for enhancements or adaptations				
	Error Analysis/ Debugging	Hours spent finding a known error in the system; may be in response to SFR, STR, SCR (includes generation and execution of tests associated with finding the error)				
	Impact Analysis/ Cost Benefit Analysis	Hours spent analyzing several alternative implementations and/or comparing their impact on schedule, cost, and ease of operation				
DESIGN	COTS/GOTS Evaluation	Hours spent in COTS/GOTS evaluation activities (i.e., identifying packages, collecting information, attending demos, evaluating and selecting COTS/GOTS packages)				
	Design Creation or Modification	Hours spent developing or changing the system, subsystem, or component design (includes PDL, design diagrams, meeting materials)				
	Design Review/ Inspection	Hours spent reading or reviewing design (includes design meetings and consultations, formal and informal reviews, walkthroughs, and inspec.)				
CODE	COTS/GOTS Integration	Hours spent integrating COTS/GOTS (may be with other software components) to produce individual applications/ subsystems (i.e., writing and debugging glueware)				
	Code Generation/ Modif.	Hours spent actually coding system components (desk & terminal dev.)				
	Code Review/ Inspection	Hours spent reading code (for any purpose other than isolation of errors) or inspecting other people's code				
	Unit Testing	Hours spent testing individual components of the system (includes writing test drivers and informal test plans)				
TEST	System Integration/ Integration Testing	Hours spent integrating components into the system or writing and executing tests that integrate system components (includes sys-test)				
	Regression Testing	Hours spent regression testing the modified system				
	Indep. Testing Support	Hours spent supporting independent testing, including training of testers				
OTHER	COTS Package Familiarization	Hours spent learning to use a COTS package (not formal training, which would be listed under training for self; also does not include evaluation)				
	Prototyping	Hours spent prototyping to investigate a particular issue				
	Training for Self	Hours spent taking courses (including computer-based training), attending seminars, etc.				
	User Support/Training	Hours spent training users and responding to their questions				
	CM	Hours spent in configuration management				
	Procurement	Hours spent procuring/purchasing, interacting with vendor regarding licensing/maintenance agreements, etc.				
	Documentation	Hours spent creating and reviewing deliverable documents				
	Management	Hours spent managing or coordinating work and reporting status				
	COTS/GOTS Other	Other COTS/GOTS specific hours not covered above				
	Other	Other hours not covered above (i.e., department and all-hands mtgs)				
	Total	Total hours per column	0.0	0.0	0.0	0.0
	Grand Total	Total hours	0.0			

Appendix D - CTIF

For Librarian's Use Only

Number:

Date:

Entered by:

Checked by:

COTS & TOOLS INFORMATION FORM (CTIF)

Use this form to obtain context and evaluation data, verify at project completion.
For each COTS product or Tool, use a separate CTIF.

Name: **Date:**

Project:

COTS Product or Tool: **Version Number:** **Vendor:**

1. Reasons for using tool or COTS: Check all that apply.

□ requirements definition □ requirements analysis □ requirements □ tracking/traceability □ design
□ simulation/modeling □ code generation □ static analysis □ compilation □ debugging
□ configuration management □ integration □ QA □ re-engineering □ testing
□ reverse engineering □ change management □ project tracking □ documentation □ training
□ information management □ reuse management □ measurement □ risk analysis □ communication
□ project planning/estimation □ application functionality

2. Support provided for tool or COTS: Check all that apply.

□ demos □ informal or partial documentation □ full documentation □ courses □ help desk

3. Activities supported by tool or COTS: Check all that apply.

□ requirements definition □ requirements analysis □ design □ coding □ testing
□ documentation □ CM □ QA □ management □ other

4. Usage frequency of tool or COTS: (Select one from choices below, enter letter of selected item here.)

a. no usage b. used once or twice c. monthly d. weekly e. daily

5. Functionality of tool or COTS: (Select one from choices below, enter letter of selected item here.)

a. no data available b. abandoned, due to lack of functionality c. major expected functions missing
d. some expected functions missing e. most expected functions present f. all expected functions present

6. Usefulness of tool or COTS: (Select one from choices below, enter letter of selected item here.)

a. no data available b. abandoned, due to problems c. many problems encountered
d. some problems encountered e. few problems encountered f. no problems encountered

7. Impact of tool or COTS on project's success: (Select one from choices below, enter letter of selected item here.)

a. impossible to estimate b. major negative impact c. some negative impact overall
d. positive & negative impacts balance out e. some positive impact f. major positive impact

8. Is COTS or tool embedded in software, i.e., is COTS being delivered as part of the system? YES □ NO □

Appendix E - Sample SEL Study Brief

Study Brief Number: 7

ISSUE: **COTS Evaluation Team**

PURPOSE: Document the SEL's understanding of the COTS Evaluation Team, for the purpose of disseminating information to the FDF community and clarification for the SEL, in regards to the COTS Study.

CURRENT UNDERSTANDING:

Team was formed in 1995 to address a move towards COTS solutions in FDD. Originally, part of Code 551, Flight Mechanics. Currently, part of the Code 550 Flight Dynamics Technical Support Office (TSO).

Who is the Evaluation Team?

- Composed of problem domain experts and mission team members, Led by Sue Hoge, GSFC analyst
- Matrixed on a as needed basis, not dedicated full-time to evaluations

What are they doing?

- Evaluate COTS for Flight Dynamics Mission Planning & Orbit Determination
- Provide evaluation services to mission teams, as requested
- Provide independent software evaluations
- Monitor new COTS products, as available/maintain data on products that meet specific domain needs
- Publish Evaluation Reports
- Update the Guidelines for Evaluating COTS at the FDF document, as needed

What process is followed?

Basic process is outlined in the Guidelines for Evaluating COTS at the FDF document

- Establish the Objectives of an Evaluation
- Establish the Evaluation Type
- Determine the Evaluation Method
    - Basic/Standard Evaluation Methods
    - Variations on the Standard Evaluation Methods
- Establish Evaluation Criteria
- Perform Evaluation
- Document Results
- Benchmarks/Regression Testing/Follow up Evaluations

What have they evaluated?

STK (AGI)
PODS (AGI)
GEODYN(Code 900, GOTS)
OASYS(ISI)
PROBE(BBN)
PATTERN (BBN)
GREAS (AGI)

What problems have been encountered?
Public awareness of Evaluation Team and services is low

What else we learned about the Evaluation Team?
They are building an "experience base" of COTS evaluations (for multiple products, multiple missions).

Guidelines document:

Domain specific, and not intended to be a general methodology for any COTS s/w evaluation
Working document with lessons learned mixed in with process
Written from a hands-on perspective

What do we suggest?
The COTS Product Evaluation Questions from SEL Packaged-Based System Development document (page 22, table 3) are valid in the COTS Evaluation Team environment. Recommend that the SEL distribute the modified COTS Product Evaluation Questions (addition of two questions suggested by Sue Hoge) as a "One Pager" to technical personnel. Recommend that the Evaluation team use modified COTS Product Evaluation Questions as part of their process, since these are issues that Sue Hoge typically addresses with Evaluation Team.

FEEDBACK: (none available at this time, email comments to responsible author)

ORIGINAL AUTHORS: Amy Parra and Steve Kraft
RESPONSIBLE AUTHOR: Amy Parra
CONTRIBUTORS: Sue Hoge

REFERENCES/RELEVANT LINKS:

- Guidelines for Evaluating COTS at the FDF document
- STK Evaluation and Test Results
- STK PODS Evaluation Final Report
- OASYS Evaluation Report
- Interview notes (from two COTS Study interviews with Sue Hoge)

HISTORY: Study Brief published 11/11/97.

Appendix F - Interview Guides

Interview Guide 1a: Initial Project Interviews

Who: project leads
Subjects covered: background and current status of project, GSS vs. MATLAB decisions, initial COTS information
Duration: 30-45 minutes

Note: This interview should also include introducing ourselves and our study to the project leads.
Interviewee:

Interviewer:

Scribe:

Date of interview:

Duration:

Location:

1. What is/are your ROLE(s) on this project (get both official titles, e.g. user, domain expert, as well as a decription, e.g. technical vs. administrative, level of involvement, etc.)?

2. What is the current status of FDSS development for this project? What are the different applications being developed? Which have begun, are in progress, or are completed? [gradually narrow down to attitude applications]

3. For each application, how is it being developed? Using GSS and UIX? Using some COTS product like MATLAB or STK? Did any modifications need to be made to the COTS or GOTS products? Describe the modifications and how they were made.

4. What deployment/development/integration process did you use to produce these applications? Where did this process come from? What process documentation or guidance did you use, if any?

5. Are you aware of the SEL PACKAGED-BASED SYSTEM DEVELOPMENT PROCESS DOCUMENT?

6. Did you follow the SEL PACKAGED-BASED SYSTEM DEVELOPMENT PROCESS DOCUMENT?

7. Is there anything that we can do to make this a more useful, easier-to-follow process?

8. How were the decisions to use these COTS and GOTS products made? What were the steps in the decision process? What were the criteria?

9. Were lessons learned recorded? Where?

10. What types of problems did you run into wi:h the COTS and GOTS products you chose?

11. What do you think are the biggest risks associated with these decisions? [try to get a mapping between the criteria mentioned in #3, and the risks mentioned here] For example:

- unacceptable performance of the application,
- reliability of COTS products,
- delays waiting for something from another group,
- delivered application is unmaintainable,
- required skills not available
- key personnel leaving or being pulled off project at crucial points
- cultural clashes between personnel from different areas
- turnaround time for error fixes or added functionality

12. Any creative ways to protect against these risks?

13. What data did you collect during the project regarding COTS?

- schedule
- cost
- errors
- standard SEL data

14. What metrics do you see as valuable in managing COTS-based projects?

15. Was there a purchasing leader for this project, who? (discuss purchasing decisions, procurement)

16. What other projects do you know are using or planning to use COTS, GOTS, or other package-based products?

17. Can I be put on your project mailing list and/or could I have access to your project Web page? What else would help me keep track of how the project is going? Where can I look at project documentation?

18. Who are the other core team members and what are their roles?

Interview Guide 2: COTS Follow-Up Interviews

Who: COTS-based project leads
Subjects covered: Follow Up COTS information
Duration: 30-45 minutes

Note: This interview should also include re-introducing ourselves and our study to the project leads.

Interviewee:

Project(s):

Interviewer:

Scribe:

Date of interview:

Duration:

Location:

Date of initial interview:

1. What did you do for the following: (try to capture the major activities, process, products, reviews)

    a. Requirements Analysis
    b. Package Identification
    c. Architecture Definition
    d. Package Selection
    e. System Integration
    f. Test
    g. Maintenance

2. What are the biggest differences between traditional development and Package-Based Development?

3. What are the advantages of Package-Based Development in comparison with traditional development?

4. What are the disadvantages of Package-Based Development in comparison with traditional development?

5 Are you familiar with the SEL Package-Based System Development Process document, Feb. 1996?

6. For an upcoming COTS-based project would you use the SEL Package-Based System Development Process?

    a. If yes, why
    b. If no, why not

7 What parts of the process and/or the document would you improve and how?

Interview Guide 3: Additional COTS Follow-Up Interviews

Who: COTS-based project leads
Subjects covered: Follow Up COTS information
Duration: 15 minutes

Note: This interview should also include re-introducing ourselves and our study to the project leads. Mention that this final interview is to verify the data we have collected, and clarify any areas on which we needed more information. For this interview, meet with the project lead and any other team members that you think would be appropriate to include, to verify all the data collected on that project.

Interviewee(s):

Project(s):

Interviewer:

Scribe:

Date of interview:

Duration:

Location:

Date of initial interview:

Date of follow- up interview:

Before the interview:
List the CTIFs that are on the Kano Drive for that project
Verify the matrix and supply any reasons why process steps were or were not followed

Bring to the interview:
Matrix for that project
Process Characterization

Actual Interview Questions:

1. Have you completed CTIFs for each COTS or Tool that you are currently using? (definitely for all SEL projects, ask non-SEL project to also comply)

If not, fill in hard copies of CTIFs during the interview with the project lead.

2. This is the process characterization that we have developed after interviewing projects. How representative is it of your project? (take notes as to areas that they believe they differ from the process characterization)

3 These are the specific process steps that we noted during interviews. (Show matrix of Steps vs. Interviews for that project) Allow me to review the data that we have from you as to whether or not you followed a certain process step. Fill in YES for project completed this step, fill in NO for project did not do this step. Give a simple reason for why the project completed or did not complete a step.

References

Condon, S., C. Seaman, V. Basili, S. Kraft, J. Kontio, and Y. Kim. "Evolving the Reuse Process at the Flight Dynamics Division (FDD) Goddard Space Flight Center." *Proceedings of the Software Engineering Workshop*, NASA/Goddard Space Flight Center, pp. 27-42, December 1996.

Glaser, B.G. and A.L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967.

SEL Recommended Approach to Software Development, Revision 3. Software Engineering Laboratory Series, SEL-81-305, June 1992

Selby, Richard, Victor R. Basili, and Terry Baker. "Cleanroom Software Development: An Empirical Evaluation." *IEEE Transactions on Software Engineering*, pp. 1027-1037, September 1987.

Waligora, S. *Packaged-Based System Development Process*. Software Engineering Laboratory, February 1996.

The Package-Based Development Process in the Flight Dynamics Division

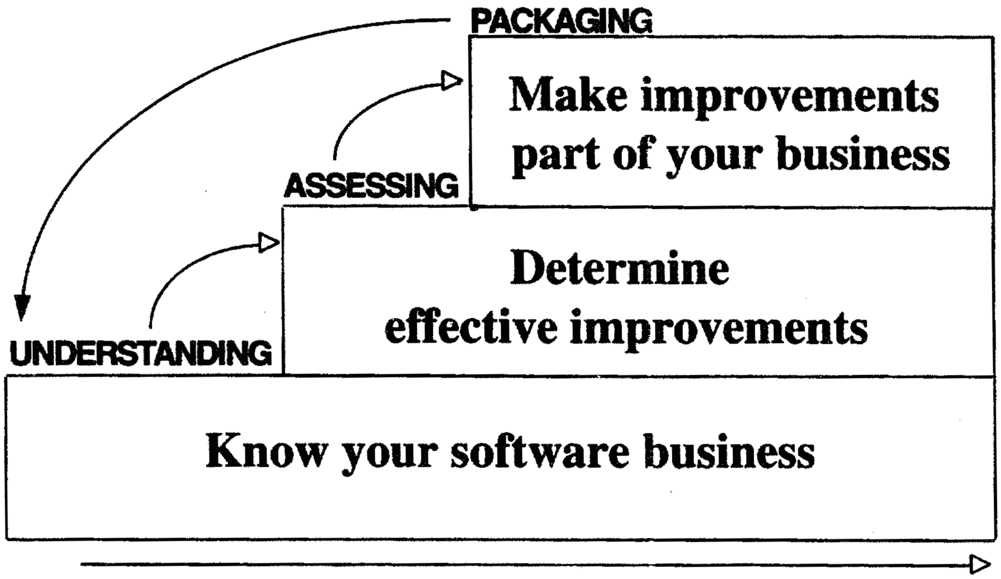
A. Parra[1]
C. Seaman[2] V. Basili[2] S. Kraft[3]
S. Condon[1] S. Burke[1] D. Yakimovich[2]

[1] CSC [2] University of MD [3] GSFC

Background

- FDD projects move to package-based
- Formation of COTS Study - Fall 1996
- Improvement Paradigm

Improvement Paradigm



PACKAGING

Make improvements part of your business

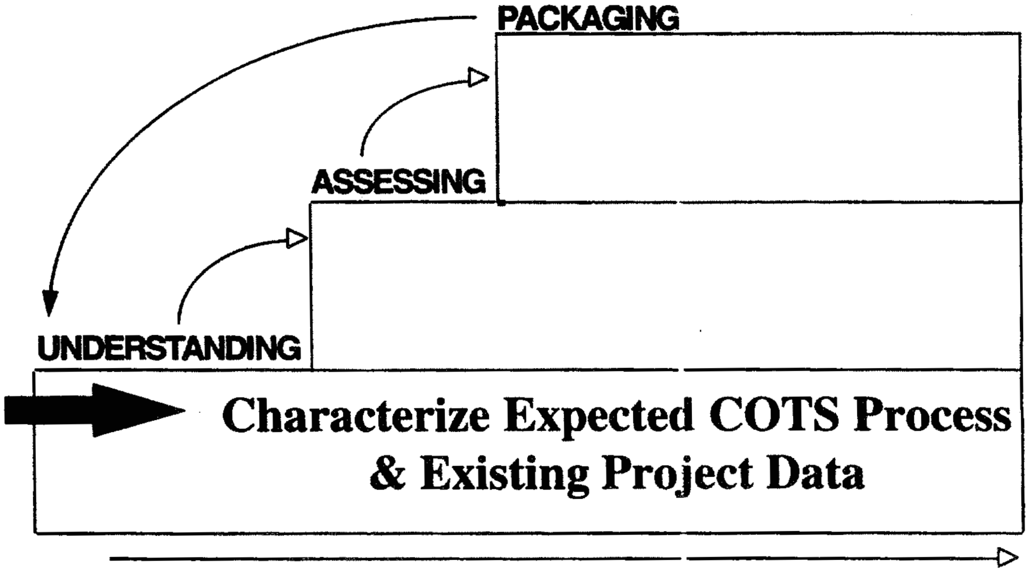
ASSESSING

Determine effective improvements

UNDERSTANDING

Know your software business

SEL COTS Study Improvement Paradigm



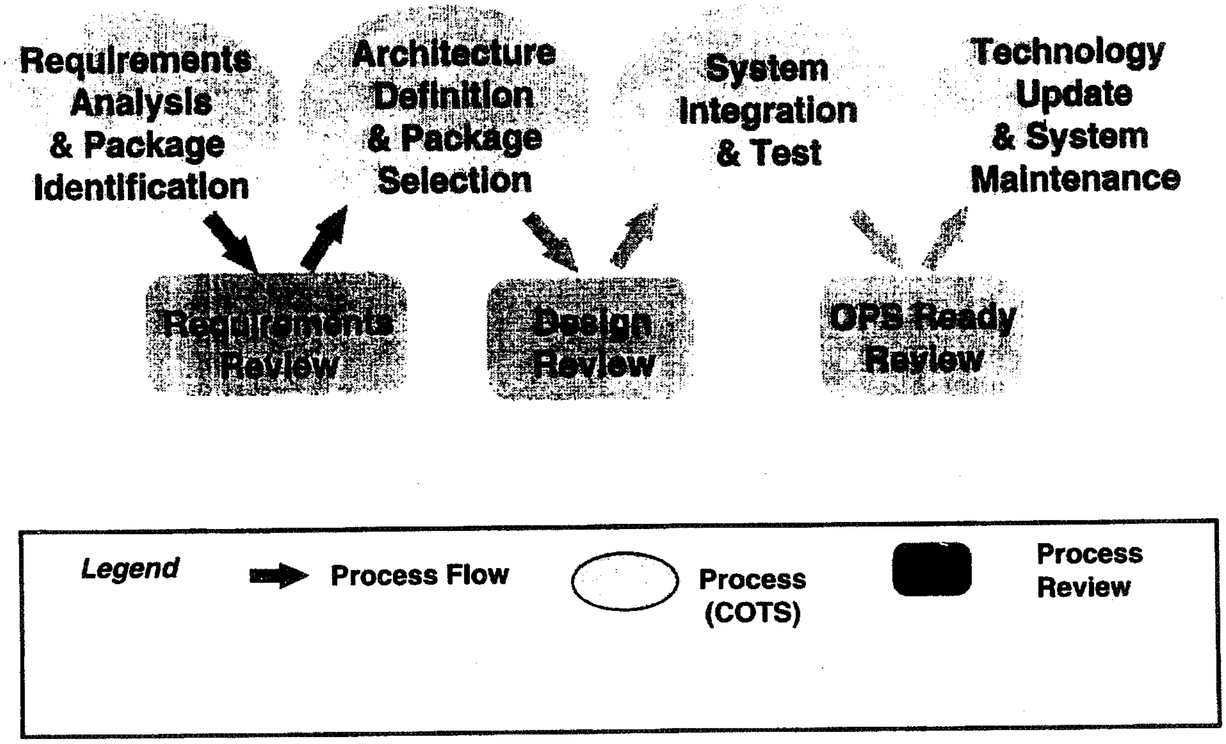
PACKAGING

ASSESSING

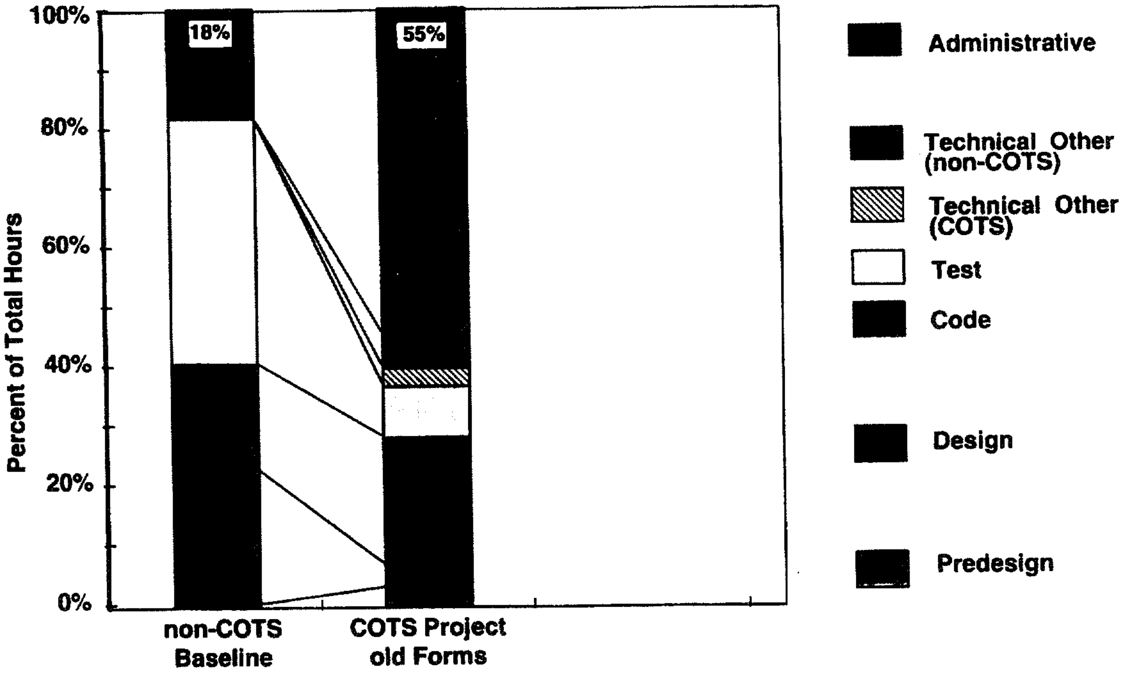
UNDERSTANDING

Characterize Expected COTS Process & Existing Project Data

Expected COTS Process

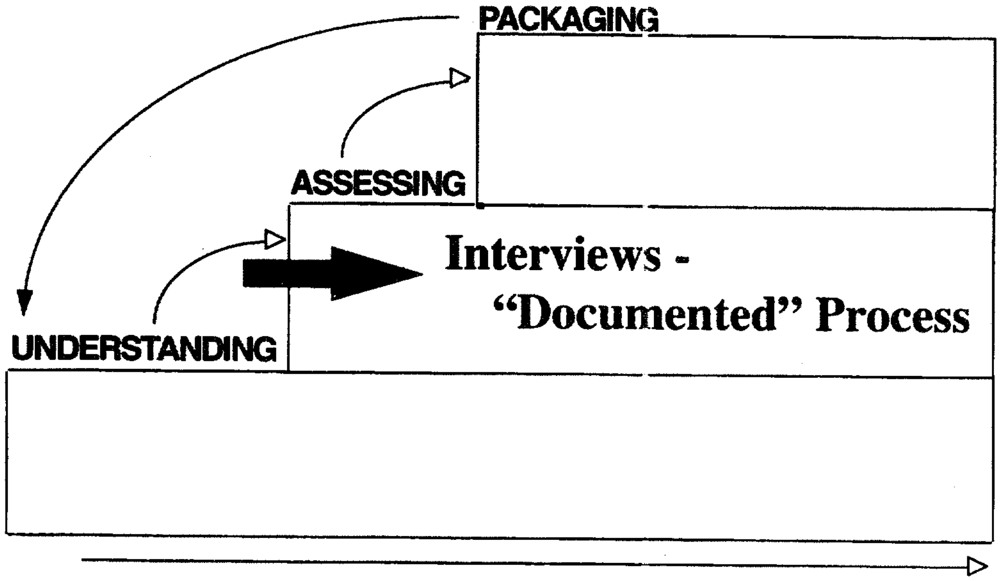

Requirements Analysis & Package Identification
Architecture Definition & Package Selection
System Integration & Test
Technology Update & System Maintenance
Requirements Review
Design Review
OPS Ready Review
Legend
Process Flow
Process (COTS)
Process Review

Existing Project Data



Data Insufficient for COTS Projects

SEL COTS Study Improvement Paradigm



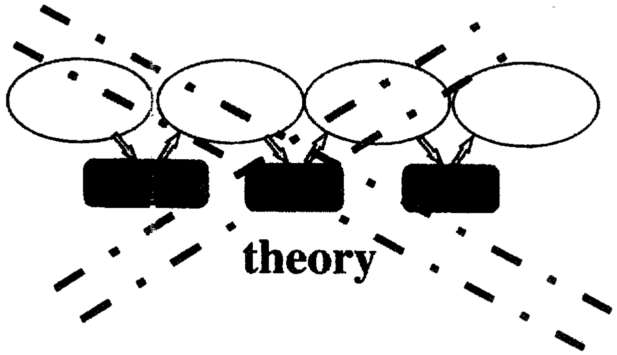
PACKAGING

ASSESSING

UNDERSTANDING

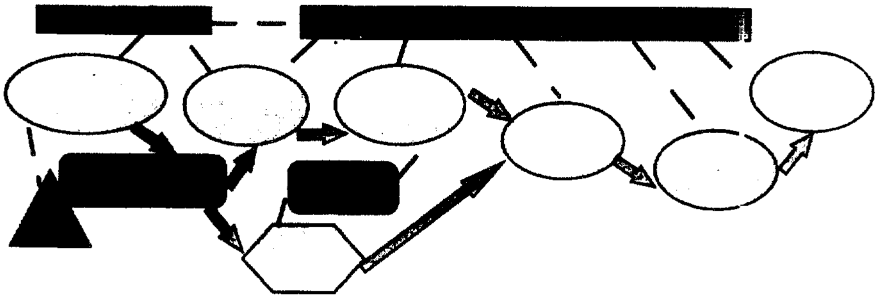
Interviews - "Documented" Process

COTS Process Comparison



theory

current FDD practice



Legend

➡ Process Flow

○ Process (COTS)

■ Process Review

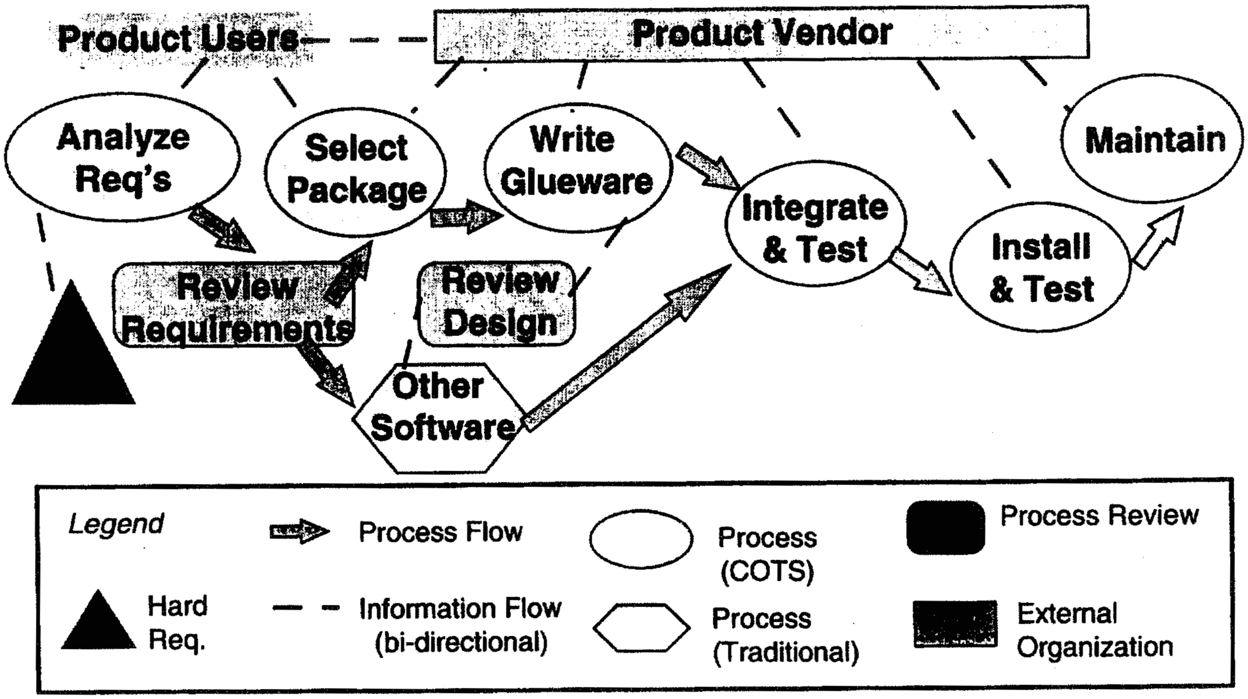
▲ Hard Req.

— — Information Flow (bi-directional)

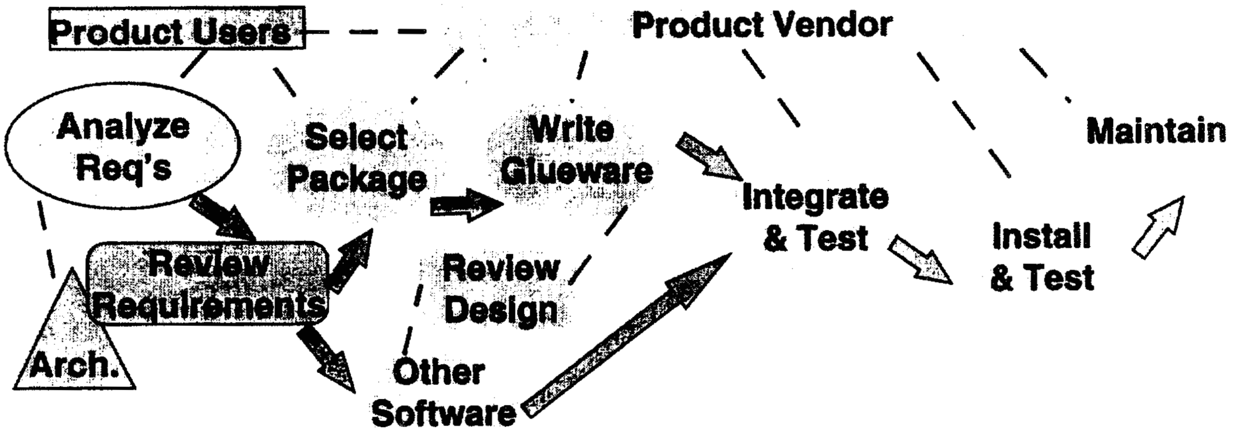
⬡ Process (Traditional)

■ External Organization

COTS Process
Current FDD Practice

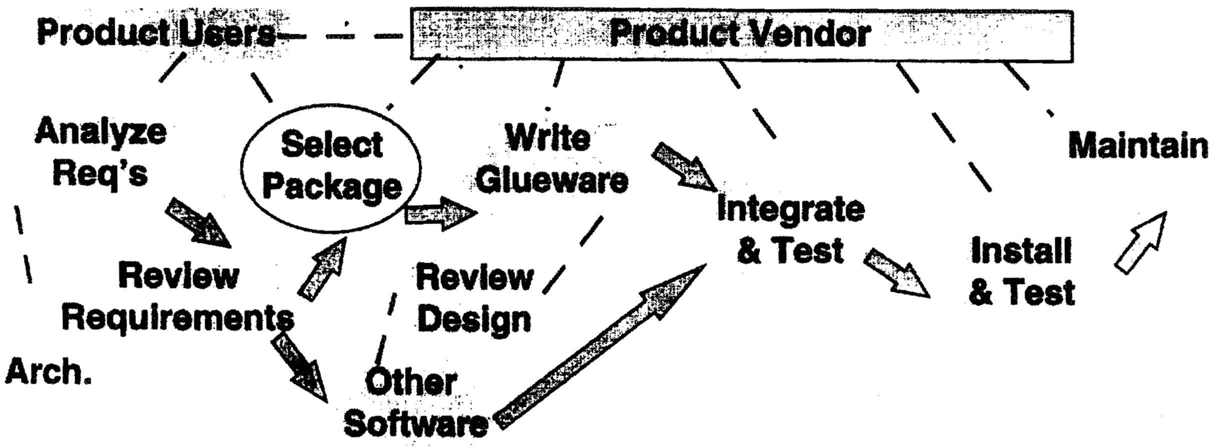


COTS Process
Current FDD Practice



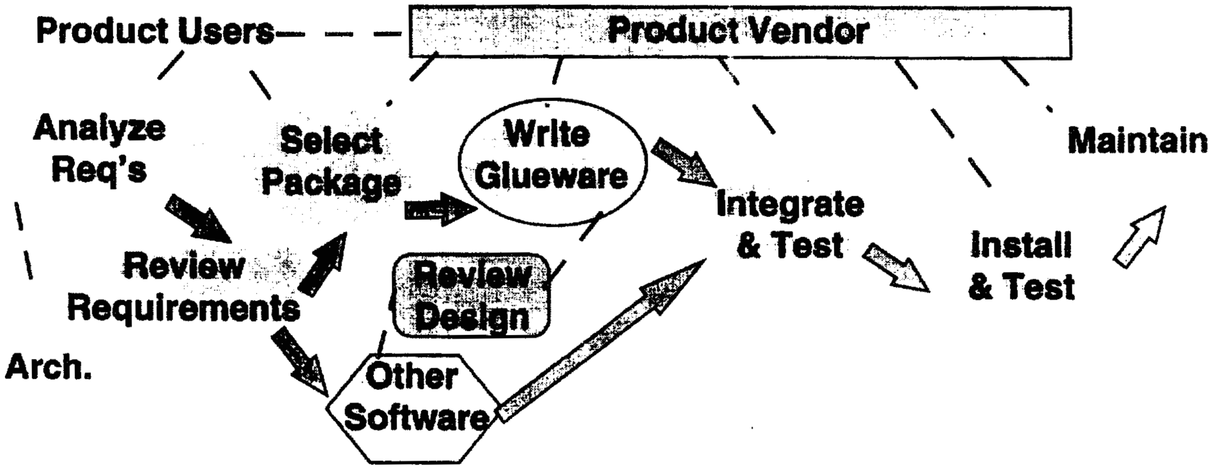
Gather External Information

COTS Process
Current FDD Practice



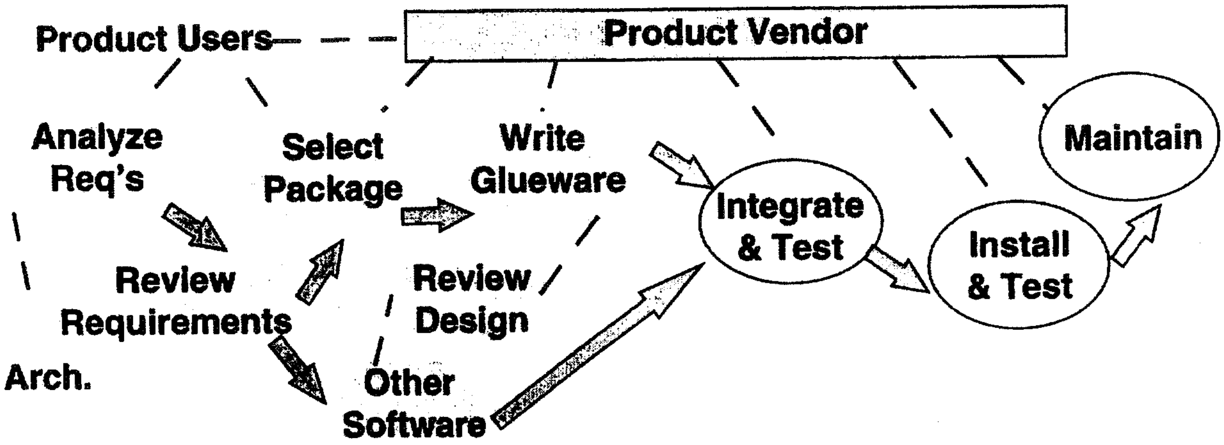
New Skills Needed: Evaluation & Procurement

COTS Process
Current FDD Practice



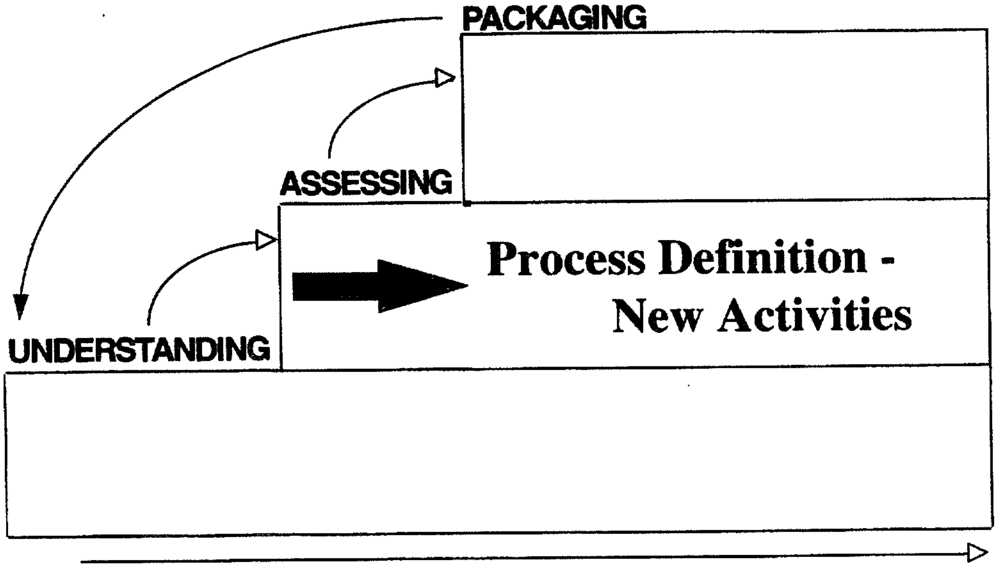
Merge COTS and Non-COTS Parts of the System

COTS Process
Current FDD Practice



Integration Process Varies (COTS dependent) & Still Interacting with Vendor

SEL COTS Study Improvement Paradigm



Process Definition - New Activities

COTS/GOTS Activities

Evaluation of COTS Packages

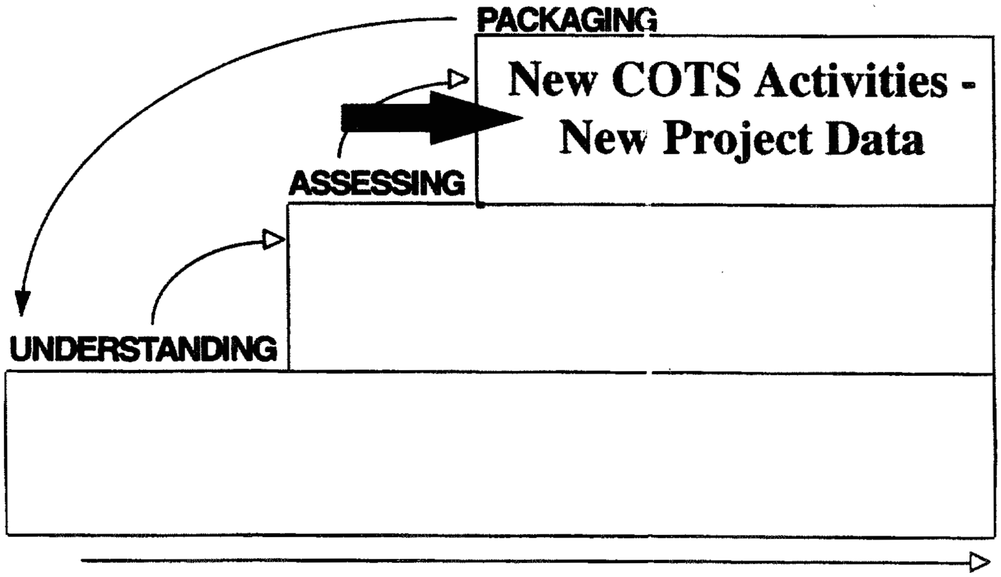
Integration of COTS Packages with Traditional S/W

COTS Package Familiarization

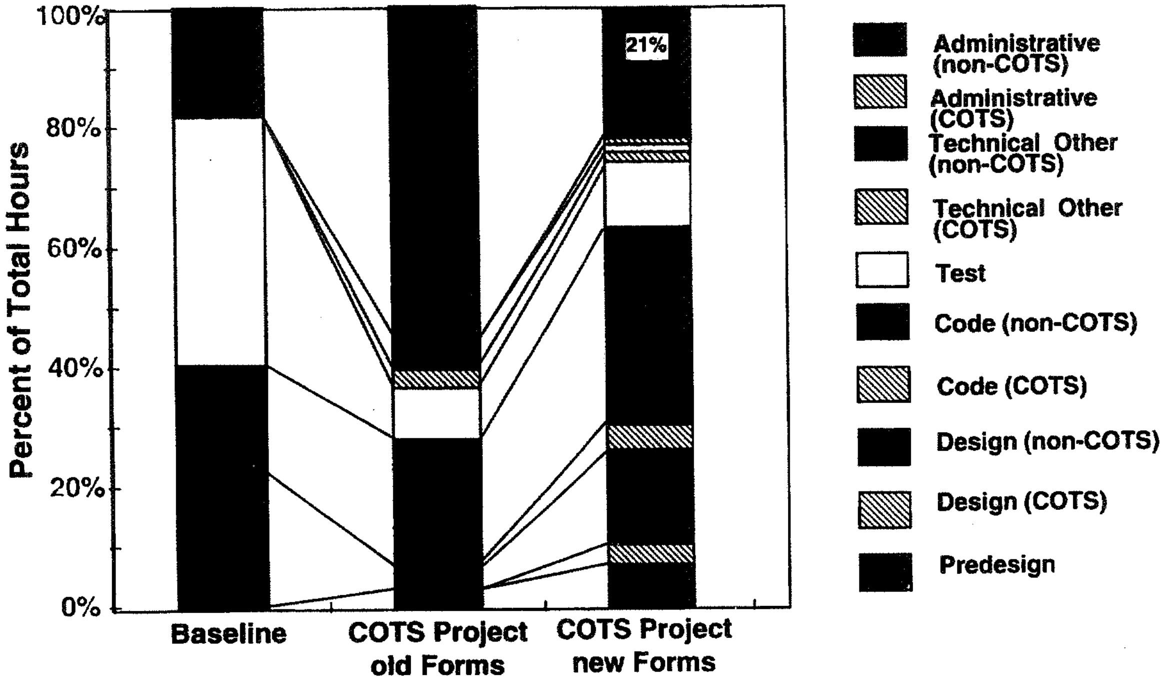
Configuration Management

Procurement

SEL COTS Study Improvement Paradigm

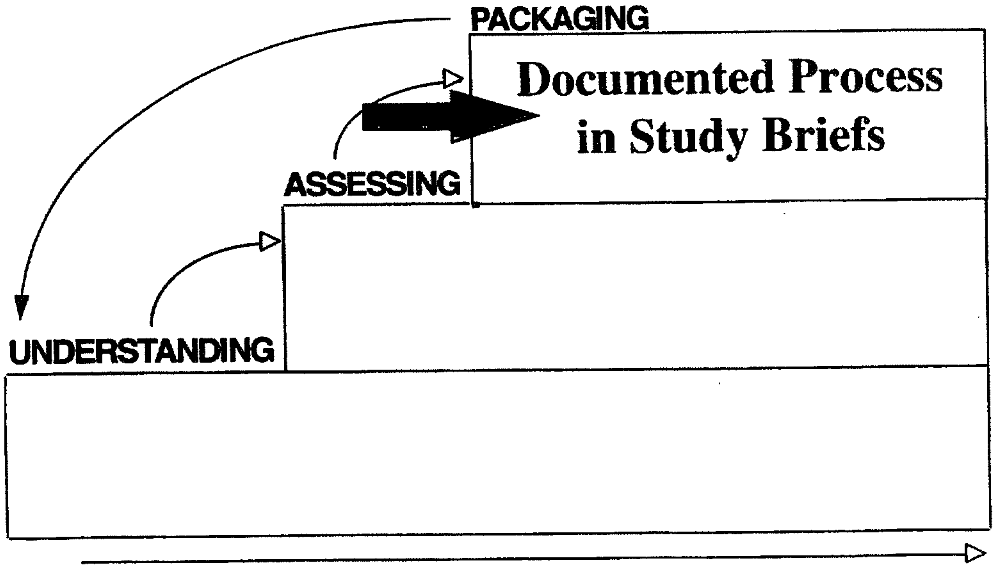


COTS Data from Projects



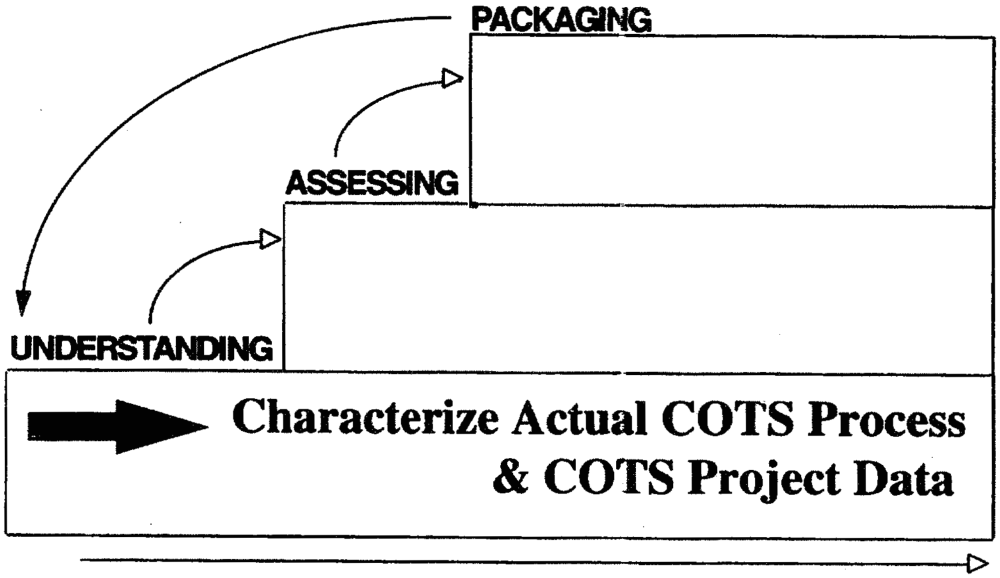
New Data Meets Needs of COTS Project

SEL COTS Study Improvement Paradigm



Documented Process in Study Briefs

SEL COTS Study
Steps to the Future



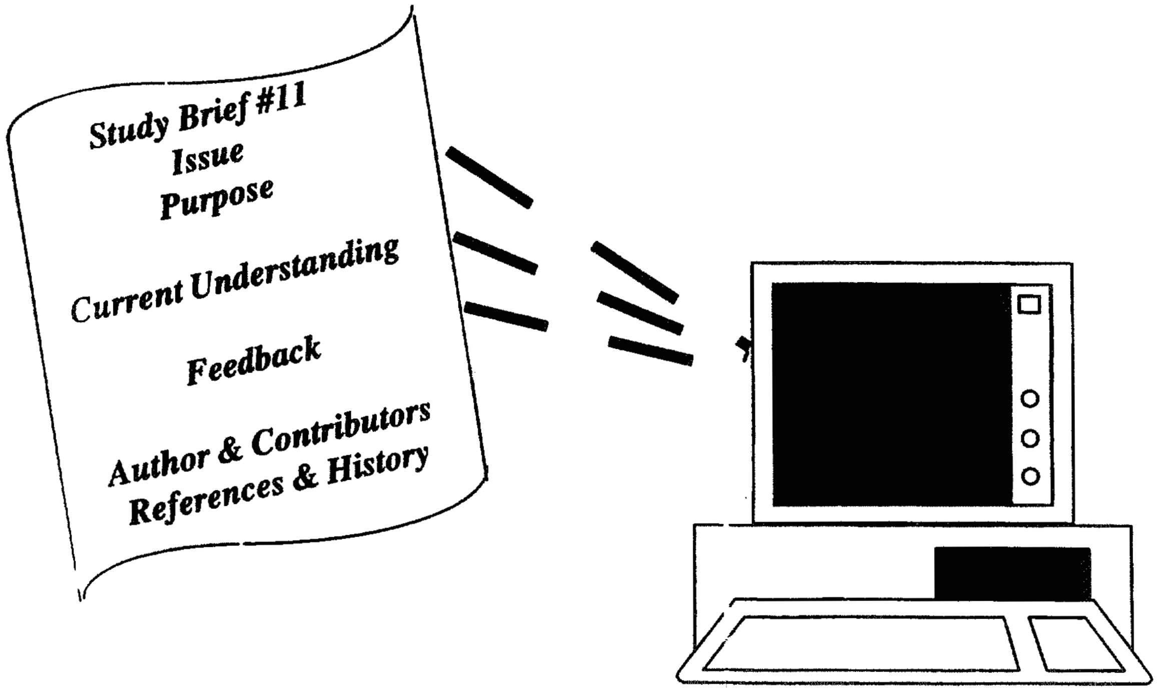
PACKAGING

ASSESSING

UNDERSTANDING

Characterize Actual COTS Process & COTS Project Data

Study Briefs



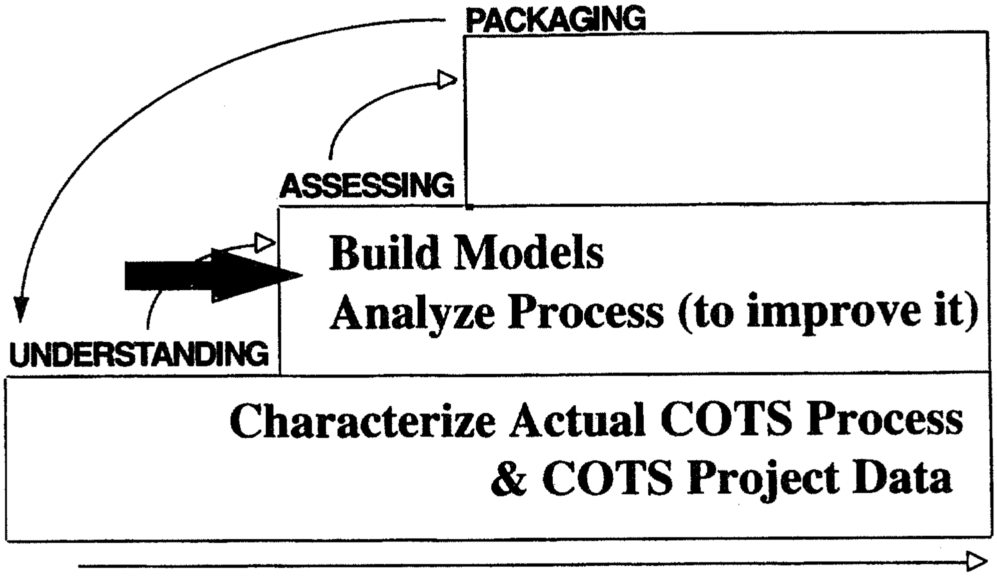
Study Brief #11
Issue
Purpose

Current Understanding

Feedback

Author & Contributors
References & History

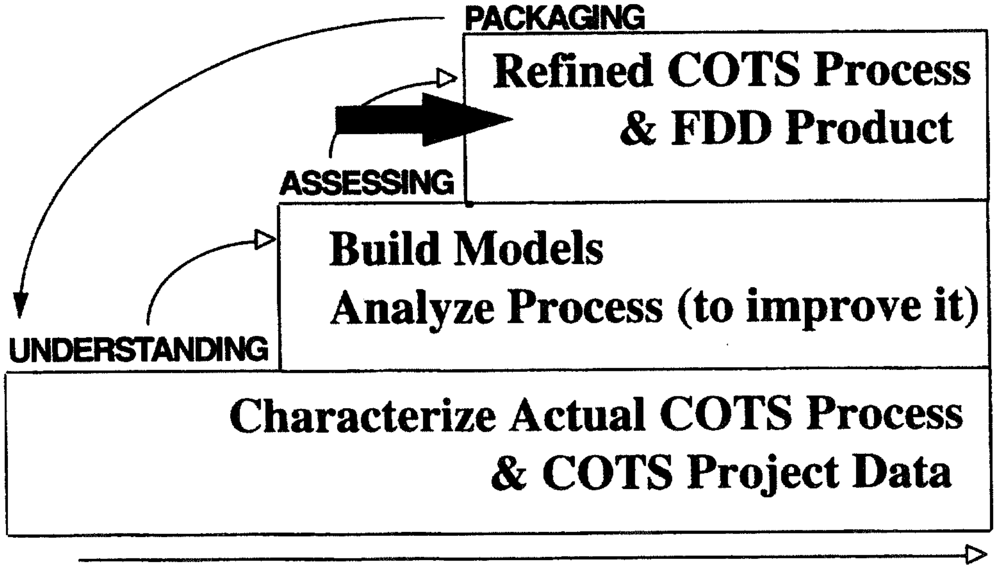
SEL COTS Study
Steps to the Future



Build Models
Analyze Process (to improve it)

Characterize Actual COTS Process
& COTS Project Data

SEL COTS Study
Steps to the Future



Refined COTS Process
& FDD Product

Build Models
Analyze Process (to improve it)

Characterize Actual COTS Process
& COTS Project Data

The Web Measurement Environment(WebME): A Tool for Combining and Modeling Distributed Data

Roseanne Tesoriero and Marvin Zelkowitz
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland USA
{roseanne, mvz}@cs.umd.edu

560791

ABSTRACT

Many organizations have incorporated data collection into their software processes for the purpose of process improvement. However, in order to improve, interpreting the data is just as important as the collection of data. With the increased presence of the Internet and the ubiquity of the World Wide Web, the potential for software processes being distributed among several physically separated locations has also grown. Because project data may be stored in multiple locations and in differing formats, obtaining and interpreting data from this type of environment becomes even more complicated. The Web Measurement Environment (WebME), a Web-based data visualization tool, is being developed to facilitate the understanding of collected data in a distributed environment. The WebME system will permit the analysis of development data in distributed, heterogeneous environments. This paper provides an overview of the system and its capabilities.

KEYWORDS

Measurement, Software development, Meta-analysis, Empirical modeling

1 INTRODUCTION

Measurement has been emphasized as an effective method for gaining control and insight into software activities. Because of this, many organizations have incorporated data collection into their software processes. However, just as important as the collection of data is the presentation, understanding, and resulting actions that accompany the data collection process. Data collection must be an active component in the development cycle of a project and not simply a passive task that results in large, mostly unused, data files.

Collection of data is inherent in the NASA Goddard Software Engineering Laboratory (SEL) [2] as part of the Quality Improvement Paradigm (QIP) [3] and as part of the Software Engineering Institute's Capability Maturity Model (CMM) [13]. However, neither activity gives much detail on how this data should be displayed. How does one view such collected data in order to present information that would be most effective to the project manager in order to aid in real-time decision making? Can we compare a new project to previously completed projects in order to determine trends and deviations from expected behavior? What do we even mean by expected behavior?

The NASA SEL had developed a tool, the Software Management Environment (SME) [6, 8], that did provide a quasi-real-time feedback on project data. The SEL has been collecting data for over 20 years on NASA flight dynamics software. Data would be entered in a data base within two or three weeks of it being collected, and then a program could be run to summarize that data for SME. Management could then use SME to display growth rates of certain project attributes (e.g., lines of code, staff hours, errors found) and compare them to previous projects with similar characteristics. This would provide two major functions: (1) Baselining capabilities so management could understand the developing characteristics of a given project, and (2) Predictive capabilities by enabling management to compare this project with previously completed projects and with idealized models of growth built into the SME system. Knowledge of software development is built into the models of SME to allow for easier analysis of collected data in the software development domain.

With the increased presence of the Internet and the World Wide Web, the nature of software development has changed. The Internet and the Web are seen now as valuable tools to be used for cooperative development in distributed environments. Recent work in the CSCW area has addressed these new requirements. Several tools have been built to automate selected distributed software processes with Web technology (e.g., software inspections [14, 12, 17], problem tracking [19, 5]). Most of this work has been focused on automating the definition and enactment of a process model. Although data measurements usually are collected automatically with the CSCW tools, the analysis of the collected data is still a mostly manual process. While the SME system was not designed to be used in a distributed, cooperative environment, we felt it provided a good basis for a more effective tool. The remainder of this paper discusses our system, called the Web Measurement Environment (WebME), which provides the same basic functions as SME, but, allowing for changing data in a distributed, cooperative environment.

2 SYSTEM ARCHITECTURE

The WebME system has a World Wide Web interface which provides a wide variety of users with access to the system and the data. For our instantiation of the WebME system, there are no restrictions to access to the system or data. However, a similar system architecture could be used within the boundaries of a corporate intranet with appropriate security measures in place.

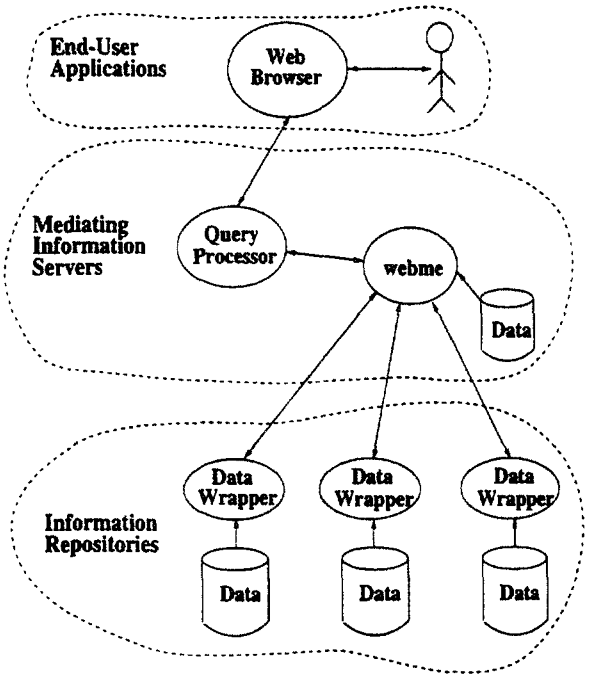


Figure 1: WebME System Architecture.

The WebME system is based on a mediator architecture [18]. A mediated architecture horizontally partitions the architecture into three layers: end-user applications, mediating information servers, and information resources. In the WebME context, the distributed databases with the software engineering data are the information resources. The *data wrappers* describe the interface between the information repositories and the mediating information server (i.e., *webme*). The Web browsers and the associated HTML forms represent the end-user application layer. The *webme* mediator is responsible fcr gathering and processing the data required to fulfill end-user requests and returning answers to the end-user.

In order to describe the system architecture, a schema of the required interfaces must be defined. Using a specialize language is a common technique used to describe a system architecture[7, 15, 11]. For WebME, we have defined a scripting language to describe the schema of the system architecture and the data definitions. In order to create the definitions for the interfaces and measurement types, an expert familiar with the development environment and databases will configure the system by creating a WebME script file using the scripting language.[1] The script will be processed into measurement class and interface definitions that will be accessible by the WebME mediator as shown in Figure 2.

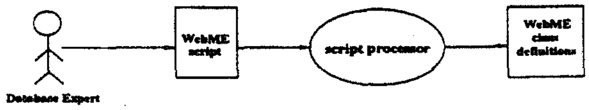


Figure 2: WebME data and interface definition process.

When an end-user makes a request, the class and interface definitions are used by the WebME mediator to gather and process the necessary data. This process is illustrated :n Figure 3.

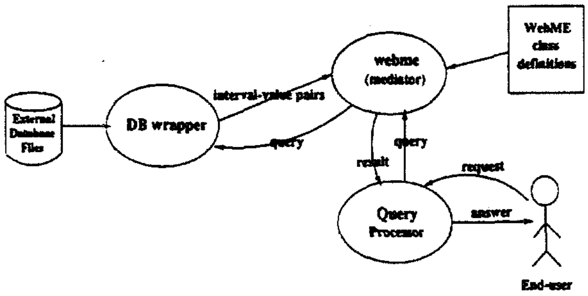


Figure 3: Using class and interface definitions.

3 COMBINING DATA

There are s:veral cases where the need to combine data from varioι s locations is necessary. For example, with a collabora:ive software development process, in order to assess and monitor the progress of the entire project, data collec:ed from each location must be combined. Another way in which data might be combined is in the graphical display. When similar data is collected from two differer t environments, it might be useful to be able

[1] Creating the script file will be described in more detail in Section 5.

to display the data on the same graph for comparison purposes.

The WebME system will use a data definition language as part of the scripting language to facilitate the combination of data in these ways. WebME will allow the user to define *classes* of measurement types, where a class will represent a given development environment, such as the NASA SEL. The measurement types (*or attributes*) represent the data collected in the development environment.

3.1 Class Definitions

Each class consists of entities that possess dimensionality attributes (using the notation of Kitchenham et al [9]). Attributes may be direct or indirect. In our context, a direct attribute is one in which the measured value for the attribute can be extracted directly from an external database. An indirect attribute derives its value from a transformation applied to other attributes (e.g., an equation).

The structural model of measurement described in [9] identifies units and values as properties of attributes. We have added an *interval* (e.g., weekly, monthly) as an additional property of the unit. A measurement instrument uses the units and the interval to supply the correct value for the attribute. The attribute definition represents the format of the data stored in the repository and is used to extract data from the external database in WebME's mediator architecture.

All data that will be displayed in the WebME system will be sequenced data with ratio scale type. For direct measures, the measurement *instrument* is an executable that will extract measured values at the desired interval from a database. For indirect measures, the measurement instrument is an *equation*. The allowable operations in the equations are the arithmetic operations (addition, subtraction, multiplication and division). The *units* and *interval* properties of indirect attributes will be inferred dimensionally from the attributes used in the equation. These indirect attribute definitions will be validated to detect invalid operations (e.g., lines of code + hours of effort is dimensionally incorrect).

In WebME, attributes are grouped into classes. Entities (e.g., software projects) are assigned to a class of attributes. Any two entities possessing the same attribute can be displayed on the same graph as long as their units are equivalent (see Section 3.2). This allows for different, but related data that are collected and stored separately to be viewed consistently. In addition, any attributes that are compatible (i.e., have equivalent units) may be plotted on the same graph.

3.2 Attribute Compatibility

The *units* and *interval* properties of the attribute definition will be used to determine compatibility for viewing and to validate the equations of indirect attributes. Two attributes are *compatible* if the units and interval properties are name equivalent. Compatible attributes may be displayed on the same graph.

The compatibility of attributes used in the equations of indirect attributes must be validated. The allowed operators are +, −, * and /. For addition and subtraction, the units and interval properties of the operands must be name equivalent. For multiplication and division, this restriction is relaxed in that the units properties may be different, but the interval properties must be name equivalent.

4 MODEL BUILDING

The consistent combination of data is one part of the problem that the WebME system attempts to address. Building meaningful models from the combined data for the purposes of process control and improvement is another.

The modeling technique used in the SME system is used to build baseline and predictive models of growth data. In 1993, a clustering algorithm using Euclidean distance was investigated [10] as an alternative to the existing SME growth models. The current growth modeling algorithms appear to be a good starting point for growth data, however, we also wanted to build baseline models for the non-cumulative raw data. This type of data is highly variable and it is often difficult to uncover trends or patterns.

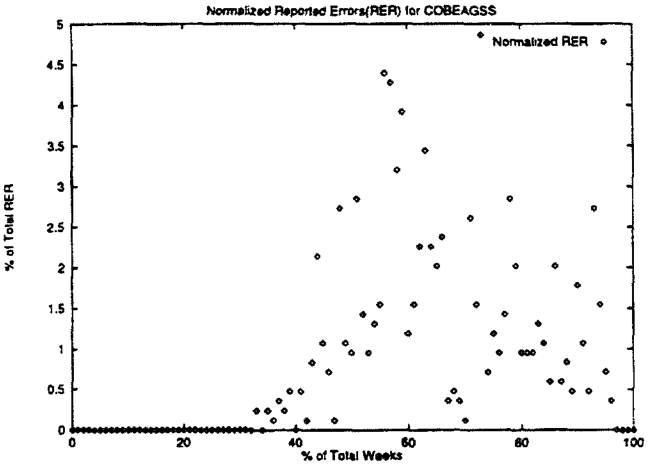


Figure 4: Scatter plot data of reported errors by week

Figure 4 represents the weekly number of error reports filed for a single NASA project.[2] It is hard to see any trend or underlying model in the data. Is there any underlying process that determines how many errors are found each week? Can we make any reasonable models of this process?

4.1 Financial models

One way to partition the data is based on using trend changes as a signal for process changes. In the financial markets, the price of a stock or commodity is highly variable. An investor's objective is to buy at a minimum price and sell at a maximum price. However, because prices fluctuate frequently, an investor would not want to trade at every trend change in the market.

The problem of trend detection for financial data turns out to be similar to our problem. We have highly variable data and we want to detect major trend changes while ignoring minor fluctuations. Techniques used to detect trend changes with financial data should be applicable to our domain.

In particular, financial markets look at long term versus short term trends. Moving averages have long been used in this domain, where an N-day moving average is the average value of some feature over the past N days. If the long term average (i.e., using a large value of N) is greater than the short term average (i.e., using a small value of N), then a stock has a decreasing trend in value; otherwise it is increasing. Such trends eliminate the daily fluctuations inherent in this form of data. If the trend moves from negative to positive, then its price has presumably reached its minimum and should be bought. If the trend moves from positive to negative, then it has peaked and should be sold since waiting will only decrease its price.

The *Moving Average Convergence/Divergence* (MACD) trading system [1] [16] determines when the long term changes in a stock's value differs from the short term changes, which signals a decision to buy or sell the stock. When the trend crosses the *signal line* (i.e., the moving average of the long term average less the short term average) in a positive direction, the price is about to rise and a stock should be bought; if it crosses the signal line in the negative direction, a sell is indicated.

4.2 Modeling Algorithm

Based on the MACD examples, we have developed an algorithm for analyzing each data attribute. Given the raw scatter plot data for some attribute (such as given in Figure 4), we want to reduce it to several linear segments that best represent the governing processes during the period represented by each segment. We will call this the *characteristic curve* and our initial goal is to find the end points for each such linear segment, which we call the *pivot points* to this curve. Once we do that, we can apply more traditional curve fitting techniques to each segment in order to develop underlying models of each process.

The three steps we have developed are:

1. Use smoothing techniques to provide a rough envelope that represents the approximate behavior of the data. This process is not sufficient by itself. For example, the data of Figure 4 results in a smoothed curve (Figure 5) which still has 12 local maxima when using an 8 point moving average.

2. Determine which of the extreme points represent a significant event for these processes. Other local maxima (or minima) are assumed to be minor perturbations in the data and are to be ignored. We call these significant trend changes *pivot points.*

3. Connect the set of pivot points into a segmented line. This represents the *characteristic curve* for the original raw data.

We outline the algorithm in the following sections:

Data Smoothing. In order to remove day to day variability in the value of a stock, N-day moving averages are used. Often a short range moving average (e.g., 30 days) is compared with a longer range moving average (e.g., 150 days) in order to compare local changes to a stock's price compared with the longer range trend. The crossover points between the short and long term moving aveiages signal trend reversals.

This *simple moving average*, however, has a weakness. If a critical point is reached (e.g., the value reaches a maximum), the damping effects of the earlier points in the average delay the signaling of this phenomenon. That is, the moving average will continue to rise for several days after the peak is reached since all points are weighted equally in computing the average. In order to enhance the perception of such directional changes, the *exponential moving average* (EMA) is used for the MACD trading system described earlier. Rather than being the simple average of the last N points, the exponential moving average is given by the equation:

$$EMA_i = (1 - \frac{2}{N+1}) * EMA_{i-1} + \frac{2}{N+1} * v_i$$

[2] All data presented here is normalized from 0% to 100%. That allows us to compare multiple projects on the same graph. The time duration for the projects considered here range from 100 to 120 weeks - about 2 years.

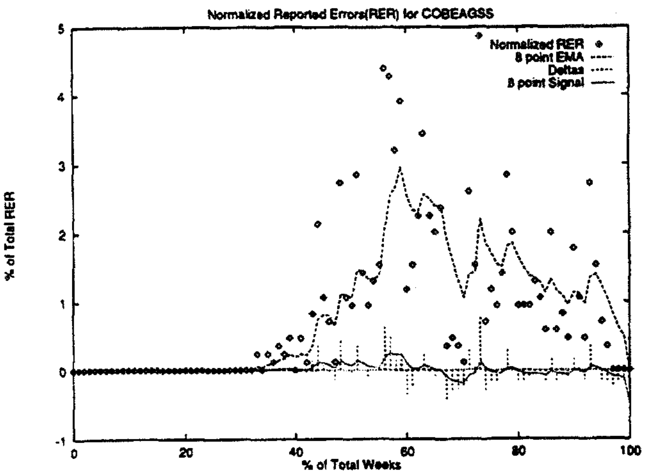


Figure 5: Smoothed data using moving averages

where:

EMA_i is the exponential moving average at time i

v_i is the new data value at time i

and $\frac{2}{N+1}$ is the smoothing constant where N is the number of points in the average.

For $N = 9$, $\frac{2}{N+1}$ has a value of .2 meaning each new point has about twice the "impact" (20% instead of 11%) that a simple moving average would have. Each successively older point has less of an effect on the total average, and the result is a moving average more sensitive to leading edge changes.

The higher curve in Figure 5 shows the effects of the EMA on the error data of Figure 4. From this EMA of the scatter plot data, we want to extract only those maxima and minima that represent significant changes in the underlying process.

Find significant trend changes. If we could simply take the derivative of this curve, we could solve for the derivative being zero in order to find the local maxima and minima. However, the actual (smoothed) data does not permit such computations. We can use the EMA to help again for this process. Between any two points we can compute the instantaneous derivative $\delta_i = \frac{\Delta v_i}{\Delta t}$. If we compute this for each time period t, and take the EMA for these delta values, we get what is called in the financial community the ***signal line*** (Figure 5). Where the signal line crosses the X-axis represents a zero EMA, or in other words, the average δ_i in the interval is 0, which represents an extreme value for the curve.[3] In our example, .his signal line crosses the X-axis 7 times. Each of these represents a critical point in the original data.

What does this signal line represent? It is the average slope of the instantaneous derivatives for the past N points. If the signal line is 0, it means that the average delta between successive points is 0 and we have a local maximum or minimum. We simply have to go back over the last N points to determine which value of time t_i represents that extreme value. We call such values ***pivot points***.

Computation of Characteristic Curve. Once we have identified the pivot points, we connect each segment with a straight line (Figure 6). This segmented line describes the general shape of the curve we are interested in. We have been able to eliminate minor hills and valleys from the curve and have left only the major features of the original data.

Figure 6 shows the characteristic curve computed by our algorithm along with the original data.

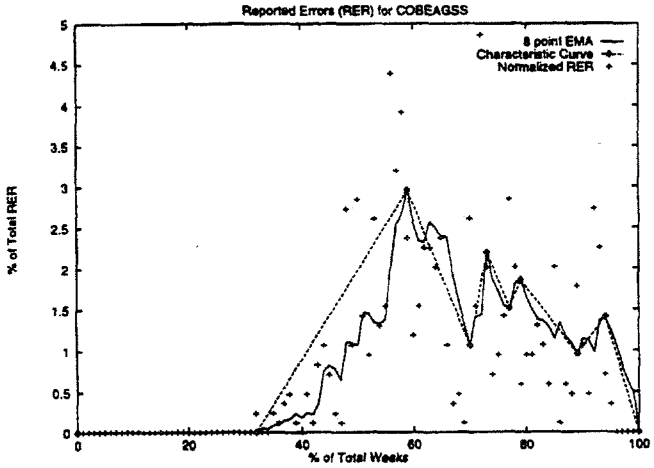


Figure 6: Raw data with its characteristic curve

One interest to us was the first dip noticed between 60% and 80% of project completion in the error data of Figure 6. Looking through old records (from 1988) we discovered that the minima point at 70% occurred just before the start of acceptance testing for the project, even though resource usage (i.e., hours worked) shows no such disruption in the process. This was also the time when the contracting organization that built the software moved into a new building. The identification of a milestone via the collected data seemed interesting, but the confounding influence of the building move concerned us. After looking at the characteristic curves of

[3] In the original MACD development, the signal line was the EMA of the difference between the long term and short term EMA. Here we are only concerned with the slope of the δ_i curve.

reported errors from other projects in this domain, we discovered similar behavior before acceptance testing.

While it appears that we can identify the start of acceptance testing (at least in this NASA environment) by the shape of the characteristic curve, we need to investigate further, the meaning of the characteristic curve. In addition, we are also working on the ability to catalog a project by the shape of the characteristic curve.

5 USING THE WEBME SYSTEM

In this section, we present examples of how the WebME scripting language is used. We only present the parts of the scripting language that are necessary here to illustrate our examples. The words in **boldface** fonts are keywords in the WebME scripting language. The words in the normal font are the parameters that are specific to the architecture being described.

5.1 Interface definitions

The interface definitions are used to describe the schema of the system architecture and the interfaces available in the architecture. Definitions of hosts (i.e., the physical location of an information repository) and wrappers (i.e., the interfaces available at the information repository) are created through the scripting language.

To define a host in WebME, the host name and port number are defined using the WebME scripting language. For example, if a *data wrapper* is listening to port number 8001 on a host named **aaron.cs.umd.edu** for WebME requests, the following statement would appear in the WebME script file:

```
create host aaron.cs.umd.edu port=8001;
```

To define a wrapper in the WebME scripting language, the host and path to the data wrapper must be identified. For example, if an executable called **getsize** which is used to extract size data from the host **aaron** is located in the **/aaron/webme/bin** directory, the following statement would be used:

```
create instrument getsize host=aaron.cs.umd.edu,
path=/aaron/webme/bin/getsize;
```

5.2 Combining data from multiple locations

To illustrate how the scripting language is used for the combination of data, consider the case of a fictitious collaborative software development process. Assume the development environment, called the Widget Development Division (WDD), has two locations where measurements of the development process for a project called *SmartWidget* are being collected and stored. For this example, the only attribute being collected is size measured in lines of code (LOC) developed each week.

If the hosts to be included in the system are *coffee.widgets.com* located in Seattle and *tea.widgets.com.uk* located in London, the hosts would be defined with the following statements:

```
create host coffee.widgets.com port=7000;
create host tea.widgets.com.uk port=8000;
```

To define the interfaces (UKsize and USsize) available at each location, the wrappers would be defined with the following statements:

```
create instrument USsize
host=coffee.widgets.com, path=/usr/bin/getattr;

create instrument UKsize
host=tea.widgets.com.uk, path=/usr/bin/getdata;
```

In addition, a class representing the WDD development environment and the attributes for size must be defined.

```
create class WDD;

/* size in lines of code for Seattle */
create attribute direct WDD.USsize
with units LOC, interval week,
instrument USsize;

/* size in lines of code for London */
create attribute direct WDD.UKsize
with units LOC, interval week,
instrument UKsize;

/* total size in lines of code (indirect attribute) */
create attribute indirect WDD.TotalSize
using USsize + UKsize;
```

Finally, the project *SmartWidget* would have to be assigned to the WDD class.

```
assign SmartWidget to WDD;
```

Now, the size of the project can be monitored at the individual site level (with the USsize or UKsize attributes) or at the entire project level (by viewing the TotalSize attribute).

5.3 Viewing compatible data

To demonstrate how the scripting language and the system can be used to display similar attributes on the same graph, suppose the attributes of reported errors (Reported), closed errors (i.e., errors that have been resolved) (Closed) and net errors (i.e., change in number of errors discovered in the current week) have been added to the WDD class with the following statements:

create attribute direct WDD.Reported
with units errors, **interval** week,
instrument getreported;

create attribute direct WDD.Closed
with units errors, **interval** week,
instrument getclosed;

create attribute indirect WDD.Net **using**
Reported - Closed;

Note that the units and interval for net errors (i.e., WDD.Net) would be inferred from the attributes in the equation. In this case, the units and interval would be errors and week, respectively.

Using the rules of compatibility described in Section 3.2, the attributes WDD.Reported, WDD.Closed and WDD.Net, could be displayed on the same graph in WebME as shown in Figure 7. However, WDD.Reported and WDD.size (as defined earlier) could not be displayed on the same graph because the units are not name equivalent.

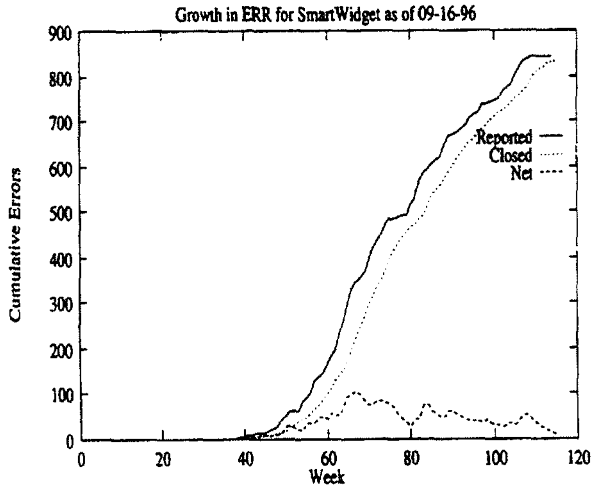


Figure 7: Compatible attributes.

Figure 7 is a graph showing the three compatible attributes using common axes. Although total errors and closed errors apparently track each other in a similar manner, the indirect attribute of open errors shows a clear bulge around week 65, which should cause management to further investigate its possible cause.

6 CURRENT STATUS AND CONCLUSIONS

The use of collected data on past projects as predictors of future project behavior is a growing phenomenon in software development. However, development environments vary widely. It is important that the baseline predictor projects have characteristics that are amenable to the new project being compared. Processes like the *Experience factory* [4] have been proposed as a means to organize such developmental practices. However, means must be found for passing information among such environments or for comparing results obtained in two different environments. A tool like WebME gives the analyst a mechanism for defining common characteristics across such domains.

At this time, the system architecture for WebME is operational allowing for access to WebME from anywhere on the WWW. The scripting language for defining interfaces and data types is being implemented. Additional data bases are under study in order to determine the effectiveness of our system in building indirect attributes across a wide range of application domains.

We still need additional experience with the proposed modeling technique before we can incorporate it into the WebME system. However, even without such additions, we have designed a system that aids software developers in accessing development data in various settings and obtaining visual feedback on the relative merits of a single project compared to a repository of related projects.

ACKNOWLEDGEMENTS

This research was supported in part by NASA grant NCC5170 to the University of Maryland. Jon Valett, formerly of NASA Goddard and now of Q-Labs, built the initial version of SME. N. Rorry Li, now of Oracle Corp., did the initial rehosting of SME to the SUN UNIX platform and added the clustering data model in 1993.

REFERENCES

[1] G. Appel and W. F. Hitschler. *Stock market trading systems*. Dow Jones-Irwin, Homewood, Illinois, 1980.

[2] V. Basili, M. Zelkowitz, F. McGarry, J. Page, S. Waligora, and R. Pajerski. SEL's software process-improvement program. *IEEE Software*, 12(6):83-87, 1995.

[3] Victor R. Basili and H. Dieter Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[4] V.R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora. The software engineering laboratory-an operational software experience factory. In *Proc. of the 14th International Conference on Software Engineering*, pages 370-381, Melbourne, Australia, May 1992.

[5] ClearDDTS.
URL: http://www.pureatria.com/ddts/ddts_main, Visited November 1997.

[6] W. Decker and J. Valett. Software management environment (SME) concepts and architecture. Technical Report SEL-89-003, SEL, January 1989.

[7] Object Management Group. The common object request broker: Architecture and specification. Technical Report 93-12-43, Object Management Group, December 1993.

[8] R. Hendrick, D. Kistler, and J. Valett. Software management environment (SME) components and algorithms. Technical Report SEL-94-001, SEL, February 1994.

[9] Barbara Kitchenham, Shari L. Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Trans. on Software Engineering*, 21(12):929–944, Dec 1995.

[10] N. R. Li and M. V. Zelkowitz. An information model for use in software management estimation and prediction. In *Second International Conference on Information and Knowledge Management*, pages 481–489, Washington DC, November 1993. ACM.

[11] J. Magee, N. Dulay, and J.Kramer. A constructive development enviroment for parallel and distributed programs. In *Proceedings of the* 2^{nd} *International Conference on Configurable Distributed Systems*. IEEE, 1994.

[12] V. Mashayekhi, B. Glamm, and J. Riedl. AISA:asynchronous inspector of software artifacts. Technical Report TR-96-022, University of Minnesota, Mar 1996.

[13] M.C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. The capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-24, CMU/SEI, 1993.

[14] J. M. Perpich, D. E. Perry, A. A. Porter, L. G. Votta, and M.W. Wade. Anywhere, anytime code inspections:using the web to remove inspection bottlenecks in large-scale software development. In *Proc. of the* 19^{th} *International Conference on Software Engineering*, pages 14–21, May 1997.

[15] J. Purtilo. The polylith software bus. *Transactions on Programming Languages*, 16(1):151–174, Jan 1994. Also available as UMIACS-TR-90-65.

[16] E. Seykota. MACD: Sweet anticipation? *Futures*, 20(4) 36+, Mar 1991.

[17] D. Tjahjono. *Exploring the effectiveness of formal technical review factors with CSRS, a collaborative software review system.* PhD thesis, Department of Information and Computer Sciences, University of Hawaii, 1996.

[18] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3) 38–48, March 1992.

[19] Web-Integrated Software Environment Home Page [1]. URL:
http://research.ivv.nasa.gov/projects/WISE/wise.html, Visited November 1997.

The Web Measurement Environment(WebME): A Tool for Combining and Modeling Distributed Data

Roseanne Tesoriero and Marvin Zelkowitz
Department of Computer Science
University of Maryland, College Park
{roseanne, mvz}@cs.umd.edu
December 3, 1997

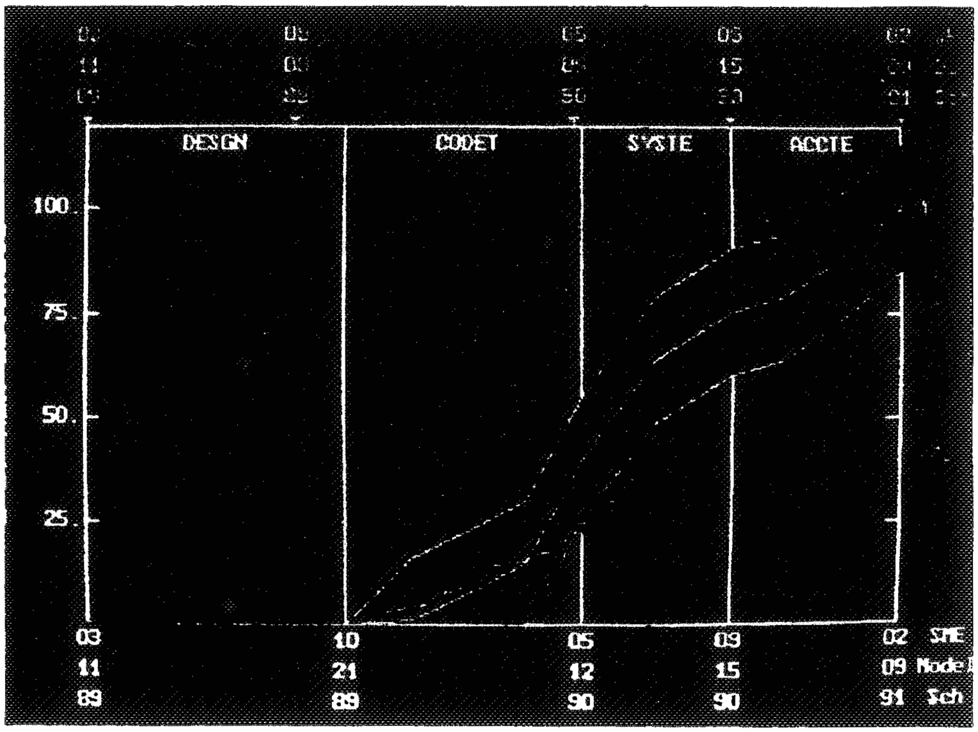
Outline

- Motivation
- Combining Data Sets
- Analyzing Data
- Status & Future Work

Software Management Environment (SME)

Provide management with feedback for ongoing projects

- Plot collected data over time (e.g., LOC, errors)
- Build baseline models for each attribute
- Compare new projects against established model
- Do "what-if" scenarios (e.g., change schedule, effort)



SME Prototype

- Prototype built in mid-80's
- Concepts not fully implemented
- Designed for SEL use at GSFC
  - DOS PC interface
  - Predefined data types
  - Predefined models

The world has changed since the mid-80s!

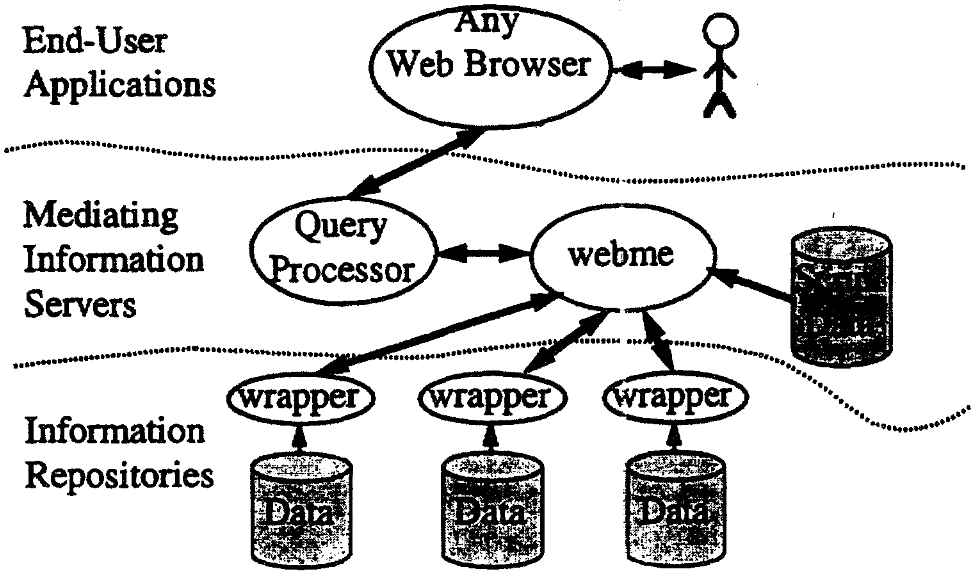
- Increased opportunity for collaborative software development
  ⇒ increased presence of the Internet and WWW
- Focus has been on process definition and enactment
  ⇒ Computer Supported Cooperative Work (CSCW) tools

Can we incorporate data analysis for project management into CSCW tools?

WebME Overview

- Goal: expand and modernize concepts prototyped in SME
  - multiple data sets
  - deeper analysis of data
- Enabling technologies
  - mediated architecture to utilize the Internet and WWW
  - schema scripting language to combine distributed data
  - new techniques (e.g., from financial domain) to analyze data

WebME Architecture



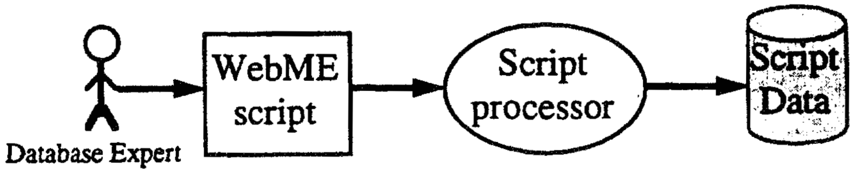
Outline

- Motivation
- Combining Data Sets
- Analyzing Data
- Status & Future Work

Scripting Language

WebME scripting language is used to define the architecture and the data of the system.

- Location: physical location of hosts with information repositories (hosts)
- Access: interfaces available at each repository (wrappers)
- Format: properties of the data in the repository (attributes)



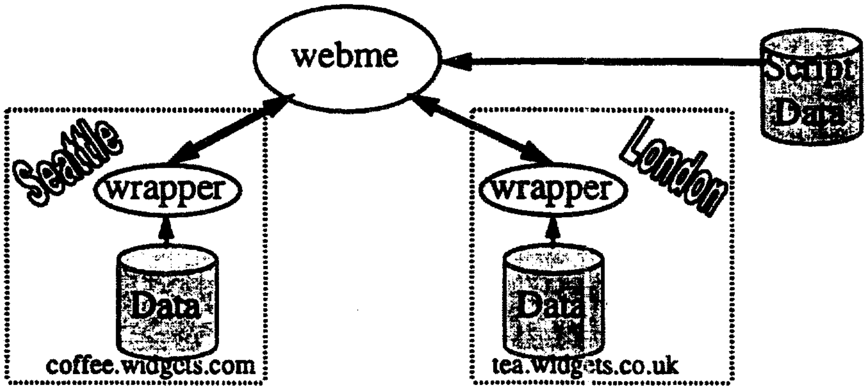
Specifying Attributes

- Direct Attributes:
  - values extracted directly from information repository using wrapper (e.g., LOC)
  - properties (units, interval)
- Derived Attributes:
  - arithmetic combination of direct attributes
  - Technical hours + Mgmt hours = Total Effort
- Combining data (meta-analysis)
  - use wrappers to get a common representation (direct)
  - apply transformations to attributes (derived)

Example:Widget Development Division

Project = SmartWidget (being developed in Seattle and London)

Attribute = size (measured in weekly lines of code)



Defining Location & Access

- Hosts (Seattle and London)

```
create host coffee.widgets.com port=2000;
create host tea.widgets.co.uk port=4000;
```

- Wrappers (USlines, UKlines)

```
create instrument USlines host=coffee.widgets.com,
path=/usr/bin/getattr;
create instrument UKlines host=tea.widgets.co.uk, path=/
usr/bin/getdata;
```

Specifying Size Attributes

- Direct Attributes:

```
/* size in lines of code for Seattle */
create attribute direct WDD.USsize with units=LOC, interval=Week,
instrument=USlines;

/* size in lines of code for London */
create attribute direct WDD.UKsize with units=LOC, interval=Week,
instrument=UKlines;
```

- Derived Attribute:

```
/* total size in lines of code */
create attribute indirect WDD.TotalSize using USsize + UKsize;
```

Combining Data Using Transformations

Error Data: Reported, Closed, Remaining

- Direct Attributes:

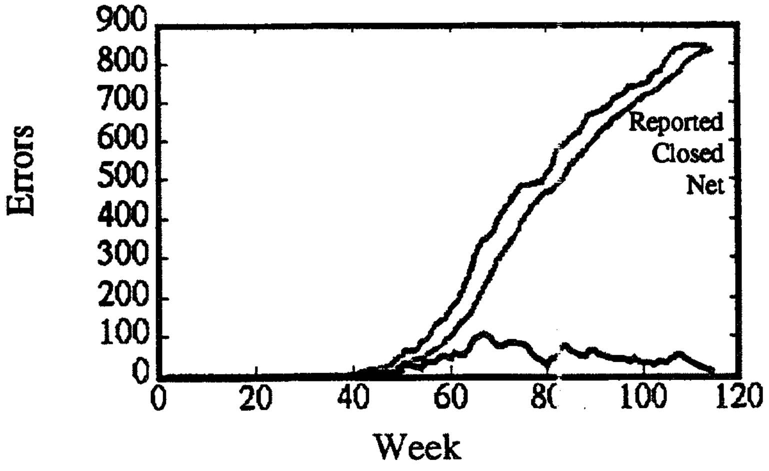
```
/* weekly reported errors */
create attribute direct WDD.Reported with units=Errors, interval=
    Week, instrument=getreported;
/* weekly closed errors */
create attribute direct WDD.Closed with units=Errors, interval=
    Week, instrument=getclosed;
```

- Derived Attribute:

```
/* weekly net errors */
create attribute indirect WDD.Net using Reported - Closed;
```

Displaying Data

Growth in Errors for SmartWidget as of 09-16-96

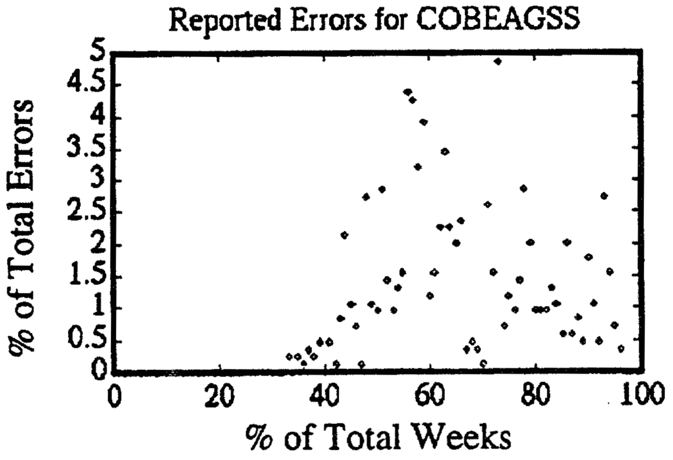


Outline

- Motivation
- Combining Data Sets
- Analyzing Data
- Status & Future Work

WebME Analysis

- Scatter plots of software engineering data are often noisy



WebME Analysis

- Objectives:
  - make sense of scatter plot data
  - automate the analysis as much as possible
  - provide deeper analysis than available with current tools
- Software engineering data has characteristics of financial data
  - stock prices are highly variable
  - stock prices follow longer trends

Can stock market analysis techniques (e.g., moving averages) be used with software engineering data?

Analysis Technique

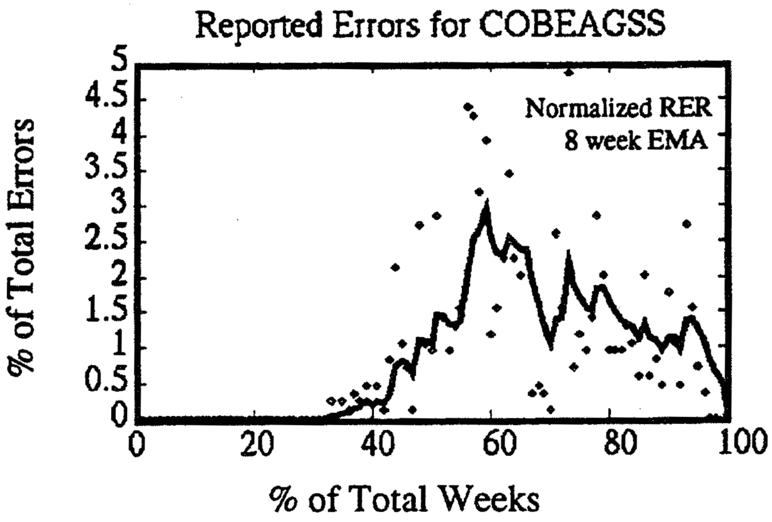
1 Smooth the data
   (using moving averages)

2 Find significant trend changes
   (called *pivot points*)

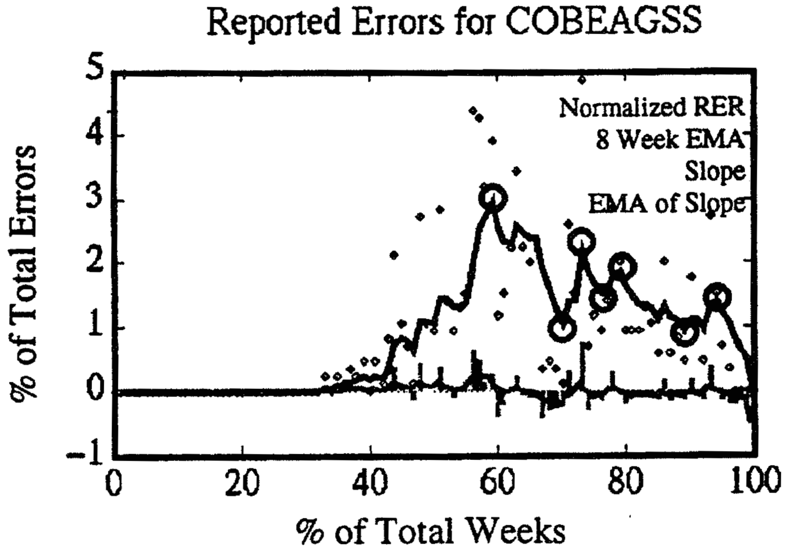
3 Connect pivot points with line segments
   (called *characteristic curve*)

4 Build a baseline model from the characteristic curves

Smooth the Data

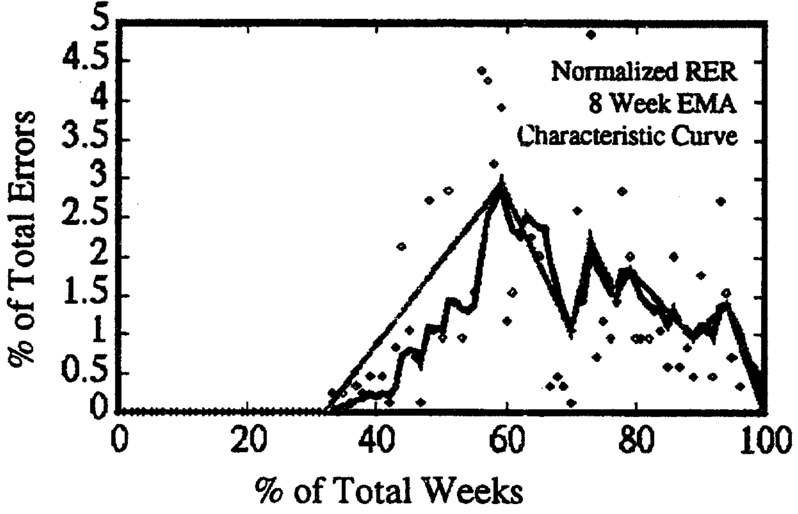


Find Pivot Points



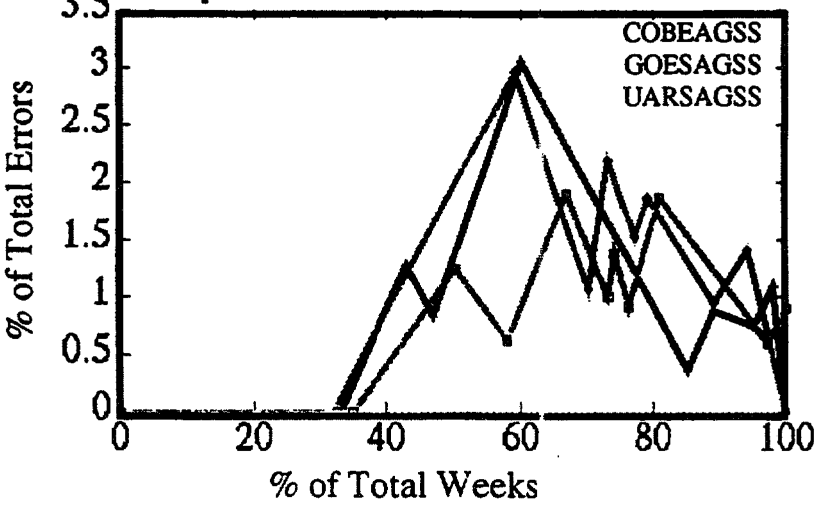
Characteristic Curve

Reported Errors for COBEAGSS

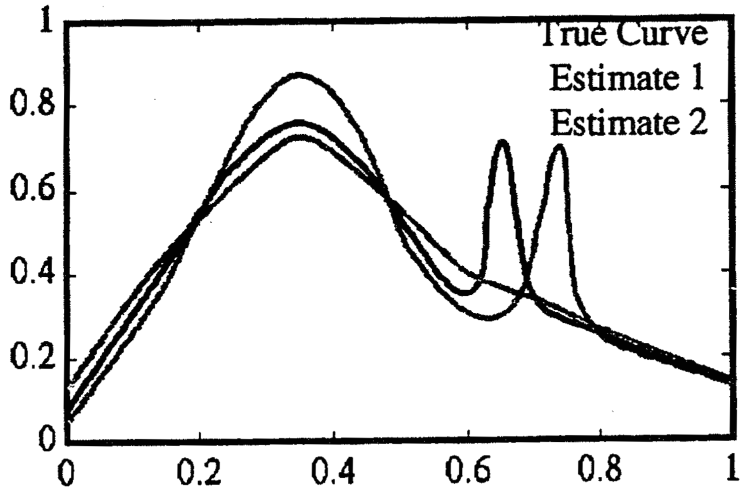


Build a Baseline

Reported Error Characteristic Curves



Capturing Visual Impression



Current Status & Future Work

- Status
  - prototype implemented
    - uses mediated architecture
    - accesses SEL data (SME data)
  - adapted financial techniques to software engineering data
- Future Work
  - implement remainder of scripting language
  - investigate techniques for building baselines
  - apply analysis technique to other databases

360823

p 18

Calibration of a COTS Integration Cost Model Using Local Project Data

Dillard Boland, Richard Coon, Kathryn Byers, and David Levitt
Computer Science Corporation
10110 Aerospace Road
Lanham-Seabrook, Maryland 20706

Abstract. The software measures and estimation techniques appropriate to a COTS integration project differ from those commonly used for custom software development. Labor and schedule estimation tools that model COTS integration are available. Like all estimation tools, they must be calibrated with the organization's local project data. This paper describes the calibration of a commercial model using data collected by the Flight Dynamics Division (FDD) of the NASA Goddard Spaceflight Center (GSFC). The model calibrated is SLIM Release 4.0 from Quantitative Software Management (QSM). By adopting the SLIM reuse model and by treating configuration parameters as lines of code, we were able to establish a consistent calibration for COTS integration projects. The paper summarizes the metrics, the calibration process and results, and the validation of the calibration.

The COTS Integration Estimation Problem. The integration of application systems from commercial off-the-shelf (COTS) products is supposed to bring the economic efficiency of component reuse on an industry-wide scale. The catch is that "some assembly is required." How much assembly will be required? How much will it cost? How does a local development organization go about answering these questions?

On one level, the answer is the same as it is for custom development: model the process, collect historical data from the local organization, and calibrate the model. But there are aspects of COTS integration that cloud the issue. The activities differ from those of custom development -- will the models and metrics used for custom development work? Will the model be adaptable to the life cycle that characterizes COTS integration? Perhaps most importantly, what metrics best capture the size of a COTS integration project? Is a line of code metric meaningful for COTS integration? Would a function-based metric be superior? Which is more useful in the estimation process?

Our Study. Our local development organization within the Flight Dynamics Division (FDD) of the NASA Goddard Spaceflight Center (GSFC) faced these questions as we transitioned from a mainframe facility dominated by custom code to an open systems environment in which COTS integration is more common. We had years of data from projects in our former environment, and a proven process for cost estimation, but we had very little that helped with effort estimation for COTS integration. When an internal Computer Sciences Corporation working group recommended the SLIM Release 4 model from Quantitative Software Management (QSM), Inc., for estimating COTS integration effort, we decided to study the model using data from our current projects.

The goals of our study were to establish

a) The usefulness of the SLIM model for estimating COTS integration effort in our environment
b) An appropriate size metric for COTS integration based on either lines of code or function points
c) A calibration and estimation process we could extended to future COTS integration projects

Because our focus was on using the recommended model, we did not evaluate the SLIM model against any other models or products.

The SLIM Model. We will discuss only the elements of the model necessary for understanding our calibration. The SLIM model is described in Measures for Excellence by Putnam and Myers (Reference 1). The SLIM software equation described in that reference can be represented as:

$\text{Product} = P \times (\text{effort}/B)^{1/3} \times (\text{Time})^{4/3}$

where

Product = size in lines of code
P = process productivity parameter
B = scale factor, a function of system size

Since the equation is non-linear, the productivity P is not a simple parameter such as lines of code per hour. For convenience the model introduces a *productivity index* PI, a dimensionless number that maps to productivity. SLIM provides a large industry data base of observed PIs by application domain. For scientific systems, the average PI is about 13. The local organization's PI is one of the critical parameters determined by calibration.

The SLIM model requires that product be measured in equivalent source lines of code (ESLOC), defined as new or modified logical source lines. These are essentially non-comment, non-blank lines. Our local standard is the delivered source instruction (DSI), which is equivalent to the SLIM definition of ESLOC. In this paper, we identify our line counts as DSI. Other measures, including function points, require a "gearing factor" for SLIM to convert them to lines of code.

The user adjusts the PI through two types of model parameters: environmental parameters and reuse parameters. Environmental parameters include staff experience, available tools, and development process. The reuse parameters address the effect of integrating unmodified components into the system. This study concerns the reuse model exclusively. We leave all the environmental factors set to "unknown" so they have no effect.

Since the definition of product counts only new and modified lines, the integration of unmodified components represents additional effort but no additional product. This is the key to understanding the SLIM reuse model, and it leads to the apparently counterintuitive result that *increasing* the amount of reused software in the system *lowers* the productivity. The model adjusts the PI downward based on the user's estimated percentage of reused software. For COTS products a code count is generally not known, so the user has to evaluate the reuse level by considering the functionality of the COTS products relative to the overall system.

The model further adjusts the PI by allowing the user to rate several other reuse parameters on a scale of 0 to 10 (or leave them set to "unknown"). The percentage of reuse modulates the effect of these settings. At the maximum reuse level and at the extreme range of all settings, we found the model lowers the PI by about 30%. The impact on schedule and effort can be considerably greater since the model is nonlinear. The reuse parameters are: research time to select product, complexity of integration with new code, analysis effort to assess impact on existing code, experience with the specific product, usefulness of documentation, relative number of functional interfaces, effectiveness of external customer support, relative percentage of functional interfaces used, level of additional documentation needed, complexity of using functional interfaces, and level of additional regression testing needed.

Projects Studied. Two recent projects in the FDD were targets of opportunity for our investigation. Both were part of a larger migration of the FDD systems from IBM mainframes to a UNIX workstation environment. These projects were the Data Collection / Data Retrieval (DCDR) system, which processes spacecraft tracking data, and the Telemetry Processing (TP) system, which prepares spacecraft attitude telemetry data for other applications to use.

These systems replaced custom FORTRAN and IBM assembly language programs with COTS products and "glueware". Development consisted of configuring the COTS products for mission support and integrating them with each other and the surrounding applications. The products used were Omega, a configurable telemetry processing engine from Veda Systems, Inc., the Oracle 7.3 family of products, and Matlab from the Mathworks, Inc., a proprietary mathematical programming language and library with built-in user interface generation capabilities. Omega required modifications by the vendor, but these were incorporated into the standard product line and are not reflected in either the labor data or technical measures included in this paper.

The architectural pattern used for the two systems is essentially the same, and is shown schematically in Figure 1. In both cases a custom-built data capture application (a "front end") provides data to Omega through socket connections. An extensive set of Omega configuration parameters, set up as part of the integration process, controls the restructuring of the data into the forms needed for input to applications. "User-supplied functions" written by developers and called by Omega supply some of the data processing capability. Omega sends processed tracking data by socket connections to glueware that loads Oracle tables corresponding to tracking data types. Applications either retrieve the data directly from the tables via SQL calls, or by invoking a "legacy data presenter" program to get the data in a legacy format. The processing for telemetry data is simpler since it does not involve the Oracle data base. A custom-built data interface utility (DIU) accepts data from Omega, completes the processing, and supplies the data to the client applications in data files. In addition to the components shown in Figure 1, scripts and graphical user interface screens help integrate the system and provide a means for user monitoring and control.

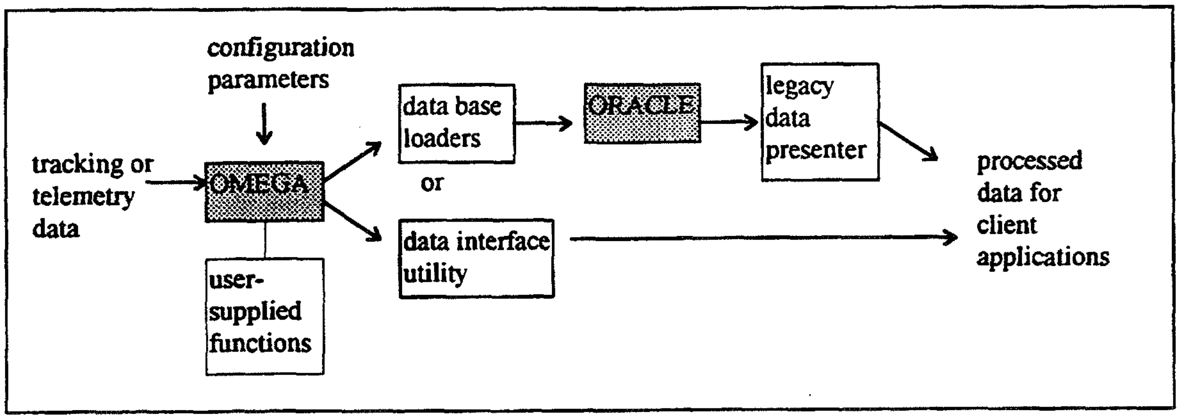


Figure 1. Architectural Pattern for DCDR and TP COTS Integration Projects
(shaded boxes represent COTS products)

The DCDR and TP systems were built simultaneously within the two-year transition from the FDD's mainframes to the new client-server system. The overall transition project was managed as a rapid rehost rather than as new development, and neither of these projects followed a waterfall life cycle. Instead, they relied on initial prototyping and successive refinement as each tracking data type or telemetry format was set up for processing. The time from the start of prototyping to full operational deployment was 22 months for DCDR and 16 months for TP.

Metrics. Table 1 presents the metrics for the DCDR and TP projects. Since our goals included the investigation of both lines of code and function points as metrics, we collected both. We made an adjustment to our standard DSI definition, however, by including configuration parameters in the count. Our reason for doing this was that the configuration parameters are in effect a language for programming Omega to unpack and convert the telemetry and tracking data. We regarded them as equivalent to lines of code, and we extended this concept to the configurable custom-built program DIU. Our DSI count includes lines of C, Pro-C, PL/SQL, scripts, Matlab, and Omega and DIU configuration parameters.

	DCDR	TP
Size in Delivered Source Instructions (DSI)		
C and ProC	16600	22136
PL/SQL	17206	
scripts	7908	
Matlab		2221
configuration parameters	4901	8010
Total DSI	46615	32367
Size in Unadjusted Function Points (UFP)		
Total UFP	1110	391
Technical Effort in Hours		
Functional Design	3035.9	2150.4
Main Build	18107.6	8779.7
Total Hours	21143.5	10930.2
Computed Productivity		
DSI/Hour	2.1	3.0
UFP/Staffmonth	8.2	5.6

Table 1. DCDR and TP Project Metrics

Our interest in using function points was primarily in learning whether the metric improved our ability to estimate the product size for a COTS integration project. In the past our organization had not used function points because they were perceived as ignoring the algorithmic complexity of scientific systems, so the function point counting process was new to us. We felt that an unadjusted function point (UFP) count would be sufficient for this first experiment with the metric. We followed the IFPUG counting rules as documented in the IFPUG Counting Practices Manual, Release 4 (Reference 2), and in Garmus and Herron (Reference 3).

Because of the nature of the mainframe-to-workstation rehost, we had no detailed specifications on which to base the UFP count. We asked the developers to perform the count based on their knowledge of the system. We provided guidance to ensure they were interpreting the counting rules consistently and from a user's perspective. We noticed that the metric hid some of the complexity that we intuitively felt drove the integration effort. For instance, because of the very large number of parameters typical of tracking data and telemetry formats, the number of data element types (DETs), a function point constituent, was often in the hundreds. Because the counting rules place logical files with more than 50 DETs into the same complexity bin, a telemetry stream with 51 parameters received the same weighting as did a stream with 250 parameters. This raised a concern for us, since we suspected the effort to configure the product was dependent on the total number of parameters to be set up.

Labor hours were collected by the Software Engineering Laboratory (SEL) data collection process. The numbers include all technical and first level managerial/administrative support effort for design, implementation, and testing up to the point of initial operational use. They do not include effort for COTS evaluation and selection, testing performed by operations personnel, or any contribution from the system engineering and project management overhead associated with the larger FDD mainframe-to-workstation transition project.

Since these projects did not follow a waterfall life cycle, we had to determine how to map the effort the life cycle phases with which the model works. The SLIM model requires that one identify relatively broad components of the effort. We worked with only the SLIM "functional design" and "main build" phases. We found we could isolate the hours associated with these by associating the functional design component with the period of the initial prototyping, and the main build component with the remainder of the schedule up to the start of full operational use.

The computed productivities are the size measure divided by the hours expended. These productivities are in line with, or are lower than, those obtained on our recent custom development projects. This is in agreement with SLIM's lowering of productivity for (uncounted) reuse. We have shown the productivities derived from the function point measures in Table 1, though we have no local historical baseline for comparison. We can only note that our UFP productivities appear to fall within normal industry ranges. We counted only function points associated with the COTS capabilities we used, not with all the capabilities the products could provide.

Calibration Process. The key to calibration is choosing those reuse model settings that seem intuitive or natural for the project, and determining by how much they lower the PI. Then one finds an "adjusted" PI by applying this delta to *offset* the reduction in productivity. Upon "playback" of the historical projects sizing data, with the adjusted PI and the reuse model settings input to the model, the model will reproduce effort and schedule observed. We refer to the adjusted PI and the reuse model settings as the calibration set.

With the calibration set one can estimate a new integration project by adjusting the reuse settings. The adjustments are made by comparing the expected project characteristics to those of the projects on which the calibration is based. There is some degree of arbitrariness to any calibration, since different combinations of model parameters can lead to the same numerical results. Therefore it is important that intended use guide the calibration. Our goal was to be able to estimate a wide range of new projects. An ideal calibration set would keep most of the model's parameters near their midrange, under the assumption that the calibration projects were average.

We used the DSI metric for calibration. To find a gearing factor for the UFP metric, we took the ratio of DSI to UFP for each project. The calibration set determined using the DSI count can then be applied to the UFP count as well. Since from the perspective of the SLIM model DSI is the more fundamental measure, this approach is consistent with the model. The UFP gearing factor is part of the calibration set. Our expectation was that, having established the gearing factor, we would be able to estimate new work directly in terms of function points for input to the model.

Calibration Results. Table 2 shows the calibration results. The most fundamental model parameter is the percentage reuse, which we decided to set to 7 (i.e., 70%) for both projects. This value is intuitive. The COTS components drove the development activities for these projects, so a reuse level greater than 50% is appropriate. At the same time, there was much custom development work in the system, so setting the reuse level near 100% seems incorrect. In this fashion we chose the range of 70% to 80%. Selecting 70% gave the best result. It let us keep most of the other model parameters set to their average value, in accordance with our plan of selecting calibration values near the midrange, and places our adjusted PI close to the SLIM industry average.

Reuse Settings (scale of 1 to 10)		
percent reuse	7 (70%)	
experience with products	2 (low)	
complexity of integration	4 - 5	
time required to select products	unknown	
all other settings average	5	
Productivity Index (PI)		
	calibration value	*SLIM database (scientific systems)*
DCDR & TP	12.9	13 +/- 3
Gearing Factors		
	calibration value	*SLIM recommendations*
DCDR	42 DSI/UFP	database: 40
TP	83 DSI/UFP	default 3GL: 80

Table 2. Calibration Results

Besides percentage reuse (a mandatory parameter), we chose to set only a few reuse model parameters different from their average values. Our guiding philosophy was to set only those parameters that we had a strong reason for setting. These included the development team's experience with the COTS products (low, since these were our first projects using these particular products) and the complexity of integration. We set the complexity a little lower for TP than for DCDR (4 versus 5) since the TP had no data base interface. We set the time required to select products to "unknown" since we had no relevant data, though the effect on the model is indistinguishable from setting it to "average".

These settings lower the PI by a total of 1.2 to 1.3 points. Our observed PI was 11.6 for DCDR and 11.7 for TP, leading us to choose an adjusted calibration PI of 12.9. With this and the reuse model settings we selected as input, the SLIM model will generate the observed PIs. The resulting calibration places our systems squarely in the midrange of SLIM's industry data base for scientific systems (PI = 13 plus or minus 3, according to Putnam and Myers).

The derived gearing factors are quite different for the two projects. We attribute that mainly to DCDR having a great deal of data base use, and TP none. In fact, referring to the table provided with the SLIM documentation, our DCDR gearing factor is very close to the default for database languages (40 logical source lines per function point), and our TP gearing factor is close to the default for general 3GL languages (80 logical source lines per function point).

Validation of the Calibration Set. To validate the results of the calibration, we investigated enhancements to the TP and DCDR that each added a thread of new data processing. These enhancements were essentially another round of development iteration for each project, performed after the systems had entered operational support. For each enhancement we made two independent size estimates, one in UFP and one in DSI (i.e., we did not use the gearing factors to predict one or the other metric). Then we adjusted the reuse factors as shown in Table 3 to reflect the level of experience of the developers and the relative complexity of the new work compared to the baseline efforts.

Adjustments to DCDR reuse settings	Adjustments to TP reuse settings
experience with product = 5	experience with product = 10
relative number of interfaces used = 2	relative number of interfaces used = 1
research time to select products = 0	research time to select products = 0
additional documentation required = 2	additional documentation required = 0
additional regression testing = 3	additional regression testing = 1

Table 3. Adjustments to Reuse Settings Used During Validation

For DCDR we estimated the change required to add a new tracking data type as 100 UFPs. We readily obtained the UFP count by going through the total function point list and identifying those elements needed to support a new data type. We treated a modified UFP as equivalent to a new UFP for this analysis. We independently estimated the volume of work as 3500 DSI through analogy with the similar data types already implemented. Table 4 shows the estimates obtained with SLIM using the adjusted reuse model settings. Constraining the model to a realistic peak staffing of 3 yielded estimates that turned out to be just slightly higher than the 3.5 months and 7 staff months subsequently observed. In contrast, the effort extrapolated from the development productivity without using the model is considerably greater than the actual effort. The reuse model appears to adequately account for changes in productivity due to increased familiarity with the COTS software; in fact a somewhat higher experience setting than the value used (5) would probably have been appropriate in this case.

	size estimate	effort predicted w/o model	SLIM predicted schedule	SLIM predicted effort
DCDR (estimate 1)	100 UFP	12.2 sm	4.05 months	8.68 sm
DCDR (estimate 2)	3500 DSI	10.2 sm	3.63 months	7.78 sm
Actual + projected schedule = 3.5 months, effort = 7 staffmonths (sm)				
TP (estimate 1)	14 UFP	2.5 sm	2.32 months	1.48 sm
TP (estimate 2)	300 DSI	0.64 sm	n/a	n/a
Actual schedule = a few days, effort < 0.2 staffmonths (sm)				

Table 4. Validation Results

When we performed a similar validation exercise for the TP, the results were inconclusive. Using the same approach as described for the DCDR estimate, we identified the unpacking of a simple telemetry type to be 14 UFP. The resulting SLIM prediction is much larger than the observed, but the reason is that the size estimate turned out to be too large. The change affected only the configuration parameters, so using a DSI estimate of 300 (the average number of configuration parameters per telemetry type) gives a better guess as to the size. This is too small a size for the SLIM model to handle, but a simple extrapolation using the development productivity yields 0.64 staff months. In fact, the actual DSI count was only 102, accounting for the even smaller effort (no more than 0.2 staff months) observed.

While it would be desirable to have more than a single validation instance, the DCDR enhancement results are encouraging. They show that the productivity adjustments stemming from the reuse settings are appropriate and that the model's behavior is consistent with our experience. The estimate is also consistent with industry norms. Figure 2 shows a report adapted from the SLIM display of the DCDR enhancement estimate for main build effort. The two calibration projects, TP and DCDR, are on the trend line for scientific systems in the SLIM data base. The prediction is within one standard deviation of the model's data average for scientific systems. The calibrated model appears to be valid in our environment, given a reliable size estimate.

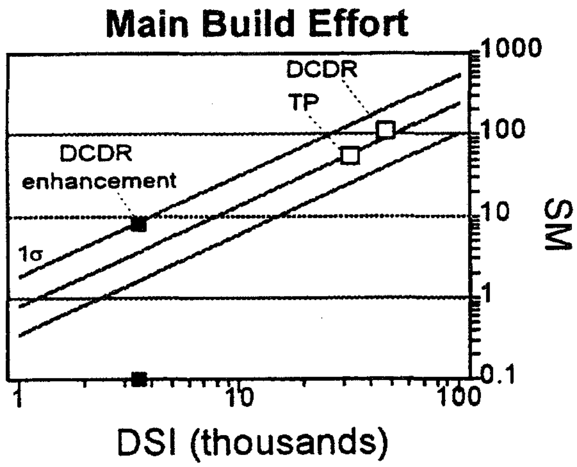


Figure 2. SLIM projection for DCDR enhancement (center line is SLIM average for scientific systems)

Arriving at a size estimate for COTS integration work is difficult, as illustrated by the TP enhancement exercise. In that case, the UFP metric, while reflecting the capabilities affected by the new data type, led to too large a size estimate. The gearing factor determined during development may be too large when applied to enhancement, or equating a modified function point element to a new one may not be valid. The TP enhancement may be characteristic of a maintenance change, while the DCDR enhancement, which did not have these problems, might be characterized as new development.

Conclusions. Our conclusions are structured around the goals of our study:

Goal: Establish the usefulness of the SLIM model for estimating COTS integration effort

Our results show that the SLIM reuse model can handle the COTS integration estimation problem in our environment. Had SLIM Release 4.0 been available to us when we began the TP and DCDR efforts, we could have obtained realistic schedule and effort estimates just by assuming the industry default productivity for scientific systems, and making some reasonable guesses as to the reuse model settings. We would of course have required good size estimates as input.

The SLIM model's use of only the new and modified lines of code, with productivity adjustments to reflect reuse, provides a straightforward way of handling the inclusion of COTS products in a system. Though it may be unsatisfying to leave as critical a modeling parameter as the percentage of reuse to guesswork, we do not see a reasonable alternative. The process worked for us. Also, even though our projects did not follow a standard waterfall life cycle, we were able to sort out the functional design and main build phases well enough to use the phase modeling in SLIM.

Goal: Establish an appropriate size metric for COTS integra ion

We were able to use both lines of code and function points to estimate the volume of COTS integration work. We had somewhat more success with the DSI metric than with the UFP metric. However, we found we had to make the definition of DSI flexible enough to count the things the developers were really manipulating during the integration process. Including configuration parameters in the DSI count achieved this for the systems we investigated.

How important are the configuration parameters? Referring back to Table 1, they account for about 10% of the DCDR lines and almost 25% of the TP lines. Without counting them, the project PIs are so different that we could not find a single, intuitive calibration set for both projects. We could calibrate them independently, each with its own characteristic PI, and this would be acceptable if our goal were project-specific cost models for the maintenance phase. Our goal is a more general model for future project estimates. Counting all the elements manipulated during integration makes the metric more consistent. Therefore we defined DSI to include configuration parameters as lines of code. They are lines of code in that they contain the programming of Omega for a specific environment and problem.

It is worth noting that the UFP count includes configuration parameters, since they appear as either external input or as data stored on internal logical files. Although the SLIM documentation recommends that only function points associated with new or modified lines of code be counted, such a breakout would have excluded the count of Omega configuration parameters. We found that a count of all function points associated with the COTS capabilities used in the applications was preferable. While our experience was that the function point counting process was labor-intensive and required quite a bit of interpretation, and that some precision was lost because of the binning of elements in the UFP count, the counting rules do provide a structured approach to the estimation process, which is valuable in and of itself.

Goal: Establish a calibration and estimation process for COTS integration projects

The calibration we performed can be extended to more COTS integration projects. In future projects for which we collect data, we will be able to collect data more in keeping with the needs of the SLIM model (for instance, planning to track effort as a function of functional design and main build), but even without this level of sophistication, calibration is possible. We also suggest following a guideline of keeping the number of calibration parameters small: focus only on the most critical of the reuse model parameters.

As we have seen, the calibrated model worked well when estimating an enhancement to the existing DCDR system. Having a model for gauging the cost of maintenance work is important, but we are aiming at more. Will our calibrated model be of any help if the products for our next system are of a different type than Omega and Oracle? As we include more projects in the calibration, the chances should improve. The estimation process will be to identify insofar as possible all the glueware and configuration parameters for the new system, and guess as to the relative percentage COTS in the system (or equivalently the relative importance of custom application components). These are the input to the calibrated model.

The early estimation problem is the most serious obstacle in COTS integration effort estimation. To see why, consider our situation at the outset of the TP and DCDR projects. It is unlikely that we could have made a good DSI or UFP count at that time, having never used these products before. How to anticipate the DSI contribution from Omega parameters? The amount of C language glueware needed? Could a reliable UFP count be based on the systems being replaced? Theoretically the UFP count is independent of implementation, but some of the major elements of our count were associated with the configuration parameters, which had no analog in the old system. Without a similar system for analogy, our only recourse would have been extrapolations based on an early product evaluation and integration prototyping -- which we were not fortunate enough to have before schedule commitments were be made.

Our overall conclusion is that COTS integration is not a fundamentally new type of development, but rather a style of development with a different emphasis than custom development. By correspondingly shifting the emphasis of the models and metrics of custom development, we can adapt and extend these methods to the COTS integration problem. We have seen how a standard custom development model can handle COTS integration by treating the products as reused components, and how the integration effort can be more fully captured by treating configuration parameters as lines of code. What we need is a structure for learning how to estimate the work volume early in a COTS integration project.

Acknowledgments. We wish to acknowledge the DCDR and TP developers Alan Stewart, Dennis Barnabie, and Alexander Calder, all of CSC, whose assistance with the collection and interpretation of development statistics was critical to this study. We also thank Steve Condon of CSC for his provision and verification of the SEL labor records, and Larry Putnam, Jr., of QSM, for his helpful advice regarding the calibration of SLIM. Finally, we express our appreciation to Dave Weidow of GSFC for sponsoring this work.

References

1. Putnam, Lawrence H., and Wayne Myers. Measures for Excellence: Reliable Software On Time, Within Budget. Yourdon Press, 1992.
2. International Function Point User's Group, Counting Practices Manual, Release 4, 1994.
3. Garmus, David, and David Herron. Measuring the Software Process: A Practical Guide to Functional Measures. Yourdon Press, 1996.

Calibrating a COTS Integration Model With Local Project Data

Dillard Boland, Richard Coon, Kathryn Byers,
and David Levitt
Computer Sciences Corporation
Lanham-Seabrook, Maryland

22nd Annual Software Engineering Workshop
December 3, 1997

dboland@cscmail.csc.com

CSC Computer Sciences Corporation
System Sciences Division

Problem: How to estimate effort required for COTS integration?

- Calibrate a cost estimation model using local project data -- but
  - How to model?
  - What to measure?
  - How to calibrate and validate?
- Our goals
  - See if a commercial model (QSM's SLIM Release 4) can model COTS integration in our environment
  - Define appropriate COTS integration metrics
  - Establish a calibration process to use in the future

CSC Computer Sciences Corporation
System Sciences Division

How to model?

- SLIM Release 4 from Quantitative Software Management (QSM)

Product = P x (effort/B)$^{1/3}$ x (Time)$^{4/3}$

Product = size in equivalent source lines of code (ESLOC)

P = process productivity parameter (presented to the user as a *productivity index* PI)

B = factor that varies with system size

- Use Delivered Source Instructions (DSI) to measure ESLOC
- Provide "gearing factor" to convert function points to lines of code

Computer Sciences Corporation
System Sciences Division

How SLIM 4.0 models reuse

- SLIM input is *new or modified* code only
- Reused software = *additional* functionality not measured by line count
- COTS = unmodified reused software
- SLIM *lowers* PI to account for COTS integration effort
- PI adjustment based on estimated percent reuse and ratings for 11 other parameters
  - complexity of integration, experience with product, number of functional interfaces, ...

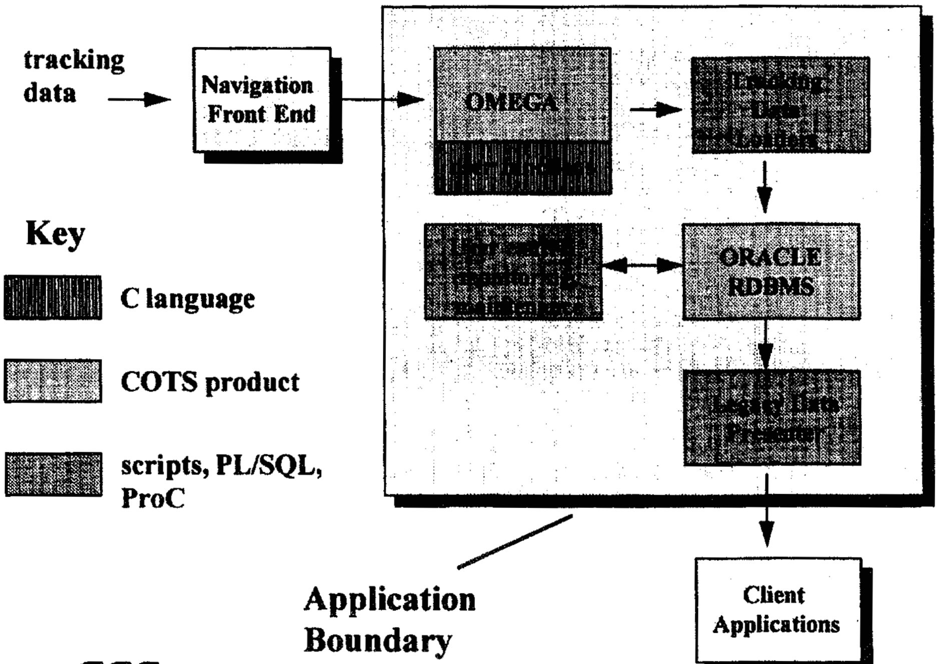
Computer Sciences Corporation
System Sciences Division

What to measure?

- **Project: "targets of opportunity" at GSFC Flight Dynamics Division**
  - Project 1: Data Collection / Data Retrieval (DCDR)
  - Project 2: Telemetry Processor (TP)
- **COTS products:**
  - Omega Telemetry Processing Engine from Veda Systems
  - Oracle RDBMS
  - Matlab from The MathWorks
- **Metrics: try DSI *and* function points**
  - Define DSI to *include* COTS configuration parameters
  - Count Unadjusted Function Points (UFP), IFPUG 1994
- **Labor hours via SEL data collection**

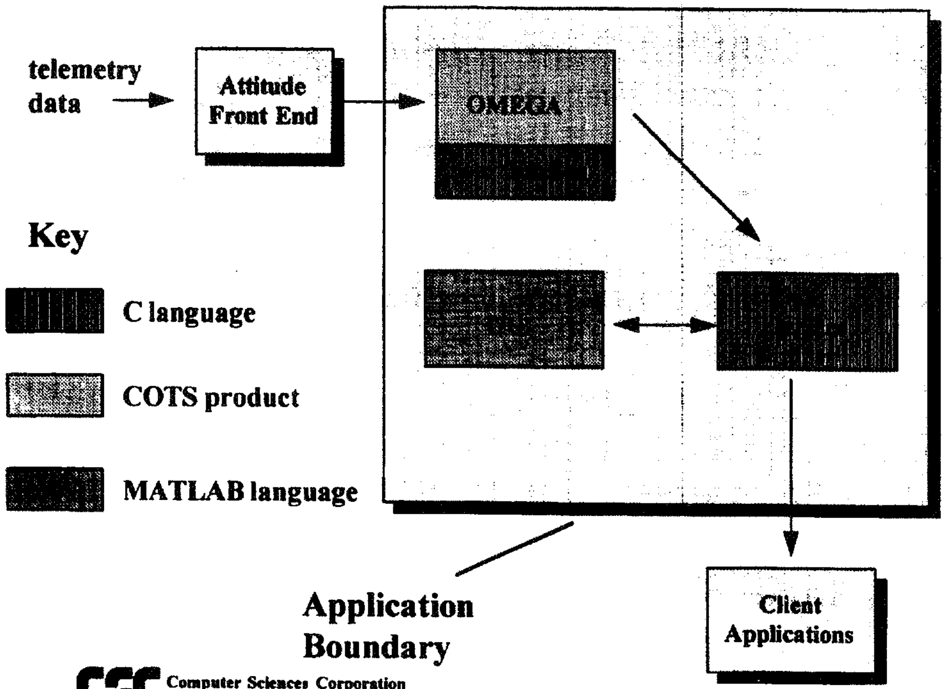
CSC Computer Sciences Corporation
System Sciences Division

Project 1: Data Collection/Data Retrieval (DCDR)



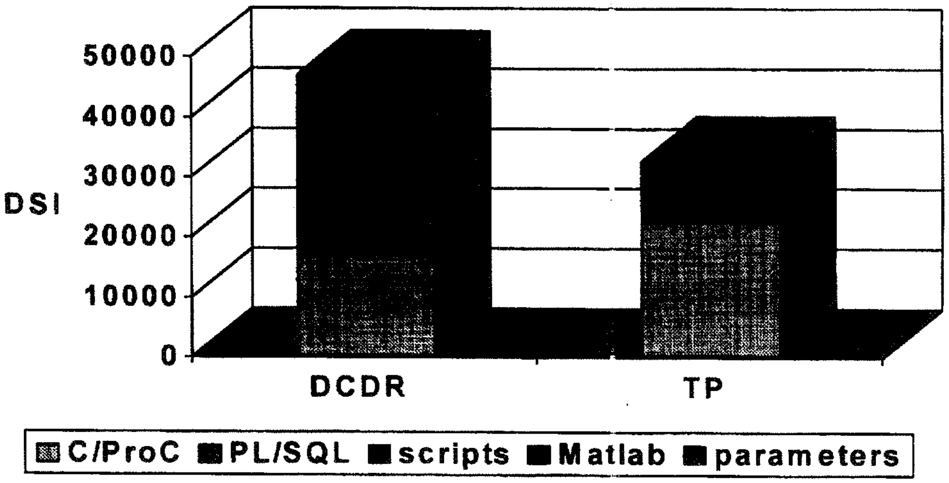
CSC Computer Sciences Corporation
System Sciences Division

Project 2: Telemetry Processor (TP)



Computer Sciences Corporation
System Sciences Division

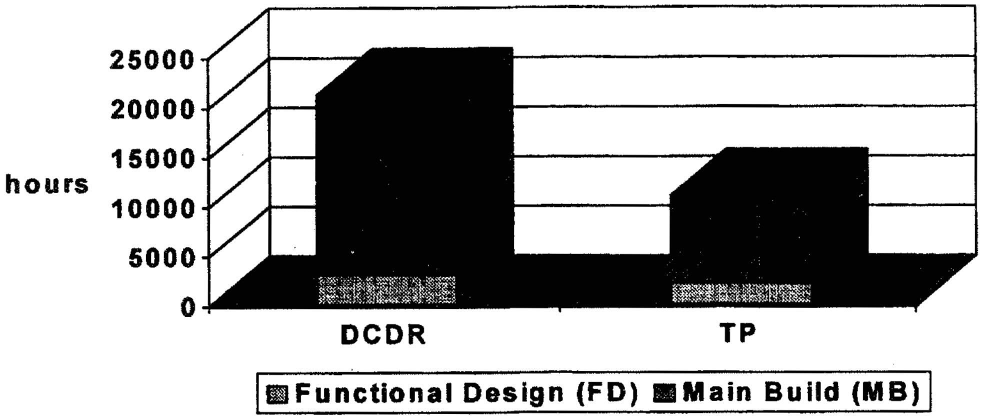
Sizing Data



UFP: DCDR 1110
TP 391

Computer Sciences Corporation
System Sciences Division

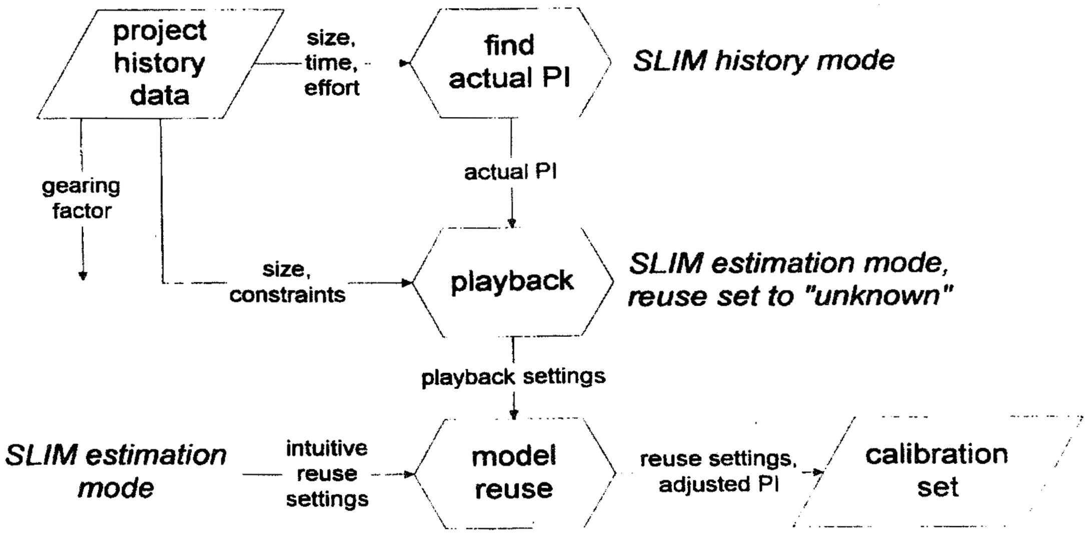
Effort Data



computed productivities:	DCDR	TP
	2.1 dsi/hr	3.0 dsi/hr
	8.2 ufp/sm	5.6 ufp/sm

Computer Sciences Corporation
System Sciences Division

How to calibrate?


project history data

size, time, effort

find actual PI

SLIM history mode

gearing factor

actual PI

size, constraints

playback

SLIM estimation mode, reuse set to "unknown"

playback settings

SLIM estimation mode

intuitive reuse settings

model reuse

reuse settings, adjusted PI

calibration set

Note: All non-reuse model parameters set to "unknown"

Computer Sciences Corporation
System Sciences Division

Calibration Example

11.6	*PI determined from development history*
1.3	*Delta to PI introduced by reuse settings*

12.9	*Adjusted PI to use along with reuse settings as calibration set*

Our SLIM calibration set

Calibration Reuse Settings (scale of 1 to 10)

percent reuse	7 (70%)
experience with products	2 (low)
complexity of integration	4 - 5
time required to select products	unknown
all other settings	5 (average)

Productivity Index (PI)

	calibration	*SLIM database*
	value	*(scientific systems)*
DCDR & TP	12.9	13 +/- 3

Gearing Factors

	calibration	*SLIM*
	value	*recommendations*
DCDR	42 dsi/ufp	database: 40
TP	83 dsi/ufp	default 3GL: 80

How to validate?

Use calibrated model with reuse adjustments to estimate TP and DCDR enhancements:

	size estimate	effort predicted w/o model	SLIM predicted schedule	SLIM predicted effort
DCDR (estimate 1)	100 ufp	12.2 sm	4.05 mn	8.68 sm
DCDR (estimate 2)	3500 dsi	10.2 sm	3.63 mn	7.78 sm

Actual + projected: schedule = 3.5 mn, effort = 8.5 sm

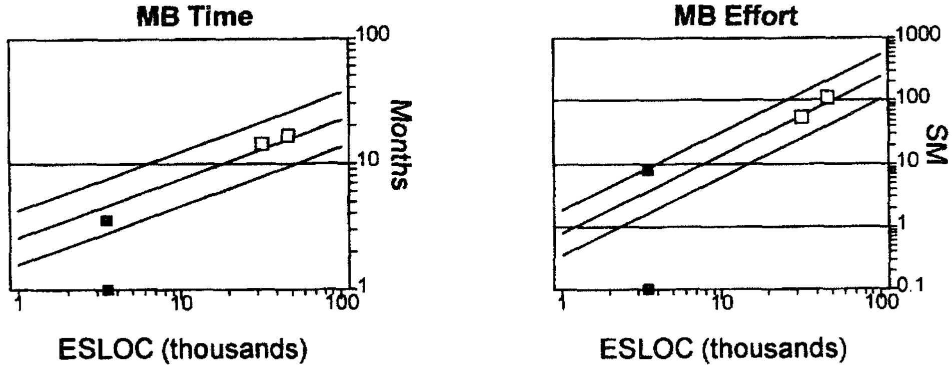
TP (estimate 1)	14 ufp	2.5 sm	2.32 mn	1.48 sm
TP (estimate 2)	300 dsi	0.64 sm	n/a	n/a

Actual: schedule = a few days, effort < 0.2 sm

ufp = unadjusted function points, dsi = delivered source instructions
mn = months, sm = staff months

CSC Computer Sciences Corporation
System Sciences Division

SLIM DCDR enhancement estimation graphs



MB			
Time	3.63	Months	
Effort	7.78	SM	Slope
Uinf Cst	0	$ 1000	Locked
Pk Staff	3.00	People	
MTTD	2.06	Days	MBI 5.2
Size	3500	ESLOC	PI 12.0

■ Current Solution □ TP_DCDR History QSM Scientific Database — Avg — 1 SD

Conclusions

MODEL

- **SLIM Release 4 reliably models COTS integration effort and schedule**

METRICS

- **DSI metric can be expanded to include COTS configuration parameters**
- **UFP metric can be used for COTS integration estimation, but needs more study**

CALIBRATION PROCESS

- **SLIM calibration gave consistent results; should be easy to expand to more projects**

CSC Computer Sciences Corporation
System Sciences Division

25-61

360834

An Architectural Approach to Building Systems from COTS Software Components[*]

Dr. Mark R. Vigder
John Dean
National Research Council
{mark.vigder | john.dean}@nrc.ca

Abstract

As software systems become increasingly complex to build developers are turning more and more to integrating pre-built components from third party developers into their systems. This use of Commercial Off-The-Shelf (COTS) software components in system construction presents new challenges to system architects and designers. This paper is an experience report that describes issues raised when integrating COTS components, outlines strategies for integration, and presents some informal rules we have developed that ease the development and maintenance of such systems.

1. Introduction

Modern software systems are becoming increasingly expensive to build and maintain and users are becoming more sophisticated in terms of the capability they expect. To build such systems, developers must use a large number of standards, protocols, technologies, and tool kits, each one of which is complex and involves a steep learning curve. Development organizations have met this challenge by using off-the-shelf software components that have been developed outside their organization and which provide much of the functionality and capability required, rather than building their own components.

Components that are bought from a third-party vendor and integrated into a system are defined as *Commercial Off-The-Shelf* (COTS) software components. Building a system from a set of COTS components introduces a different set of problems than building a system from scratch or building a system by re-using components that have been previously constructed internally in the development organization [5,12]. Many of these problems are introduced because of the nature of COTS components: they are truly black-box and the developers have no method of looking inside the box; developers have little or no influence over the maintenance and evolution of the components; and the behaviour of the component may be inadequately specified to understand its behaviour in a multi-component system. Often the COTS component is meant to run as a standalone application and has no mechanism for interacting with other programs.

[*] NRC Report Number 40221

In order to address these problems we have been experimenting with building systems by integrating COTS components. Among the objectives of these experiments is to look at: technologies that support component integration such as CORBA or ActiveX; languages that are useful for gluing components together [2,8,14,15,17]; and system architectures for using COTS components [3,6,10,11]. We are looking at these problems from the perspective of the integrator using COTS components rather then from the perspective of the builder of the COTS components. This paper is an experience report that describes issues raised when integrating COTS components, outlines a software architecture for integration, and presents some informal rules we have developed that ease the development and maintenance of such systems.

2. Building Systems from COTS Software

COTS software is a software component that a developer acquires from a third-party and integrates into their system. The COTS component supplier has the component ready-built and is supplying and supporting the identical component for numerous customers.

Developers have been using COTS components for many years [4,6,7,9...]. Traditional COTS components include operating systems, databases, and procedural libraries. Newer examples of COTS components include: complete stand-alone software systems that are being extended or integrated with other applications; components built using the emerging component standards such as ActiveX, JavaBeans, or CORBA; and application frameworks that a developer can tailor using inheritance or plug-ins.

The characteristics of COTS software that makes the software development process different from using custom-built components are:

- Developers do not have access to source code. Because they do not have access to source code developers cannot modify the code to change the functionality of the component (perhaps a good thing!). It also means that analysis, instrumentation, and testing of the component must be done in a totally black box manner.
- The component user has little or no control over the evolution of the system. The system developer who is integrating COTS components is simply one of many customers to the COTS vendor. The developer does not control, and may have minimal influence, over how the component evolves. The functionality added to each update of the component may not be as required by the developer, it may not be ported to the platforms the developer requires, it may interfere with the operation of some required functionality, or it may interact in some unexpected way with other components.
- Complete and correct behavioural specifications are not available. The specifications provided for COTS components are not always correct nor complete. Even if the COTS vendor provides a functional description, this does not always satisfy the needs of the integrator who may need to know more detailed behavioural specifications and resource requirements of the COTS component. Integrators may use COTS components in ways not foreseen by the COTS vendor.
- Sets of COTS components may be mismatched. The mismatch between components can arise for many reasons

[6] such as the data model, functional mismatch, resource clash, or process model used. Sometimes the mismatches are not found until quite late in the development process.

- Many COTS components are designed as standalone applications and may not easily interact with other COTS or developmental software.

A COTS based development is fundamentally a problem of integrating black-box components rather than building components. This integration process is not easy [6,12]. It is error prone, requires significant amount of coding, and is difficult to test and debug. In addition, many COTS components have a high level of volatility. Commercial components are often subject to frequent upgrades. These upgrades may not have the added functionality/bug fixes desired by the integrator. Critical functionality which existed in a previous version may have been removed in a subsequent upgrade. In some cases the integrator may wish to substitute similar components from different vendors in new releases of the system.

In order to be able to deal with the construction problems there are a number of properties that are desirable for an architecture that integrates COTS components.

- Plug-and-play of components. The architecture of the system must allow the substitution of components. Component substitution can involve substituting one version of a component for a different version, or substituting a component with similar functionality from a different vendor.

- Decoupling between components. There must be minimal coupling between components. Coupling can involve both functional coupling, such as procedure calls, as well as other dependencies such as resource contention or architectural assumptions. The architecture must allow for the isolation of components.

- Hiding unwanted functionality. In order to differentiate their product from competitors, COTS vendors often overload their systems with a large amount of functionality. Far from being an advantage in the COTS based system, the system architect may wish to remove this functionality. Since this cannot be done with a COTS component, the architecture must provide designers with a mechanism for masking the unwanted functionality so that it is inaccessible to the end-users and/or the system programmers.

- Debugging and testing. Since COTS components are black-box it is impossible to access their internals for the purposes of testing or debugging. An architecture and design cannot eliminate this problem, but it can include the capability of monitoring and verifying component behaviour during runtime, and preventing faults in a component from propagating through the system.

3. Example of COTS Integration

In order to better understand the issues relating to building system from COTS components we have conducted a number of experiments in building such systems. The most significant one that we have undertaken, and the one used in this paper to illustrate the architectural issues, is a distributed imagery management system. The capabilities of the system include the following:

- The system is capable of storing and retrieving various types of media (photographs, video, sound, etc.) Some of these artifacts will be stored in digital format while others will be stored as physical artifacts in a library.

- Artifacts will be stored at sites that are physically distributed. Some of the artifacts are replicated at different sites.
- A catalogue will be available on-line of all the artifacts. Each distributed site will have its own catalogue of locally held artifacts.
- Users will be able to electronically order physical copies of the artifacts, to download artifacts that are digitally available, and to replicate catalogue records when artifacts are downloaded or ordered.
- Special hardware/software (scanners, digital cameras, etc.) will be provided at workstations to generate digital artifacts and to store and catalogue them for later retrieval. Other workstations will be used for product preparation. These workstations will access on-line artifacts and process them to create products such as pamphlets, brochures, multimedia displays, etc.

The physical layout is shown in Figure 1. The system is constructed using a client server model. Server sites contain the cataloguing information and the artifacts in digital form. Clients communicate with the server in order to store, catalogue, search and retrieve artifacts. Servers communicate with each other to replicate artifacts and database records. All communication in the system is through standard internet protocols. Clients access the server through web interfaces.

3.1 Catalogue Server Architecture

The catalogue server stores all the cataloguing information for locally held artifacts. The architecture of the servers are shown in Figure 2. Requests are received from the clients by the web server and passed on to the glue components. The glue invokes local components to perform functions such as image conversion and database storage.

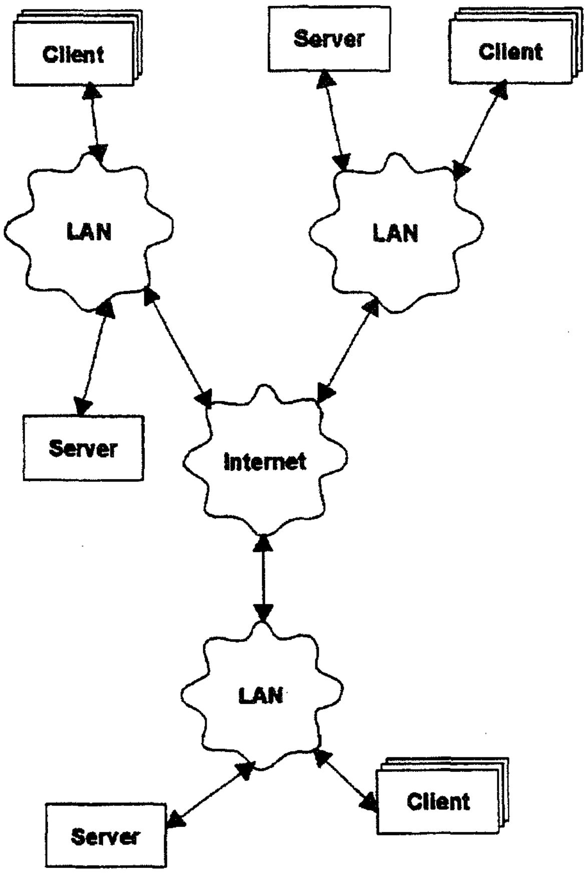


Figure 1.
Client-server model.

The various servers of the system have similar functionality but very different performance and resource requirements. One server is the main repository and contains replicas of most of the artifacts. This server is available to the entire user community. Other servers will be running on local desktop machines and be used by only two or three people within the department. The server architecture, and as many components as possible, should be portable across this wide range of platforms.

The COTS components that have been used to build the servers include the following:

- Databases. The database must be ODBC (or eventually JDBC) compliant. This allows us to use desktop databases such as Microsoft Access in the local server setups while substituting enterprise databases such as Oracle or Sybase in the main repository. We then can use identical code on both servers for database access.
- ActiveX components. We use ActiveX components to perform some well defined operations such as protocol implementation and simple image manipulation. Using ActiveX restricts us to the Win32 platform. However, by appropriately wrapping these components we hope to be able to configure the system for different platforms by substituting the appropriate platform specific components. This will

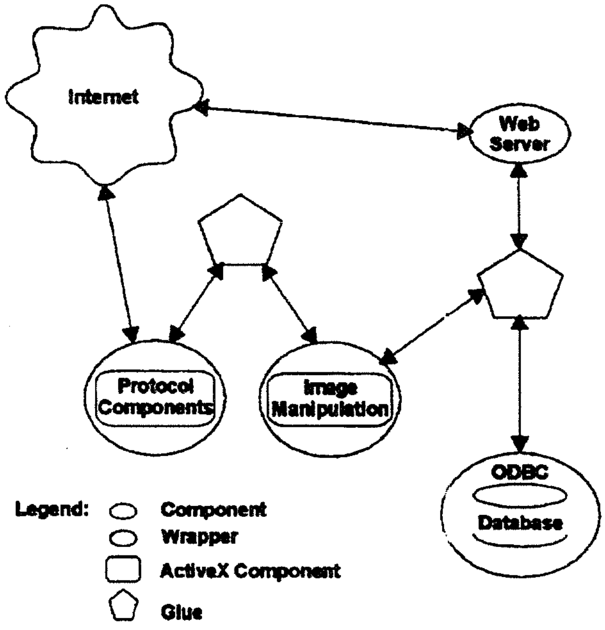


Figure 2.
Server side architecture.

require little or no changes to the other software components of the system.

- Object libraries. We use object libraries to build much of the middleware, particularly the CGI scripts. Most of the middleware is written in Perl so the object libraries are not strictly COTS as we do have access to the source and they are distributed under the Gnu license. However we have treated these in a manner similar to COTS components.
- Web servers. We have restricted ourselves to standard HTTP and CGI in order to be compatible with any of the web servers currently on the market.

Two principles we have tried to follow in the design of the system, and which will be discussed further in Section 4, are the use of wrappers and glue. Wrappers are code that we design and implement and provide the only allowed access method to the wrapped component. Glue is the middleware that manages the integration of the components. The wrappers surrounding components and the glue integrating components are shown in the server architecture of Figure 2.

Access to the COTS components is accomplished through open and standard interfaces where such standards exist. This includes standard Web protocols, such as HTTP and CGI, and standard APIs such as ODBC. Where such a standard does not exist, as in the case of many of the ActiveX components, a wrapper is built around the component with an interface that we can control.

The wrappers and glue on the servers have been written primarily in Perl, with some use of Visual Basic. By using Perl (and using it carefully) much of the middleware glue can be ported to new platforms simply by copying the scripts and configuring the server appropriately.

3.2 Client Architecture

A basic client consists of a standard web browser. Through the use of HTML forms this allows for searching and retrieving information from the database, submitting work orders, etc. For this type of client the only commercial component required is the web browser and the underlying infrastructure (operating system).

Further COTS component integration will be required for clients that have specialized hardware and/or software. This includes clients that create and catalogue artifacts or production workstations where a set of artifacts is processed into a final product. For example, a production workstation will require a software application to process images and generate reports and brochures. Numerous applications exist for this purpose and one (or more) can be used on the workstation. We do not restrict which application a client should use.

Users should be able to interact with the servers and with the local software packages in an integrated and seamless manner. For example, once a set of items has been located by a catalogue search the user should be able to download these artifacts directly into a project in the product preparation application.

An example architecture for a client is shown in Figure 3. There are two major commercial components to the system: a web browser; and an application for developing the product. The user, through the browser interface, searches for the desired artifacts which may be stored locally or remotely. Once they are found, the artifacts are downloaded into the product preparation application. The user then interacts with this application's interface to generate the required product.

Integration code is responsible for receiving the artifacts from the server, opening a project in the preparation application, and adding the downloaded artifacts to this project. A wrapper is placed around the

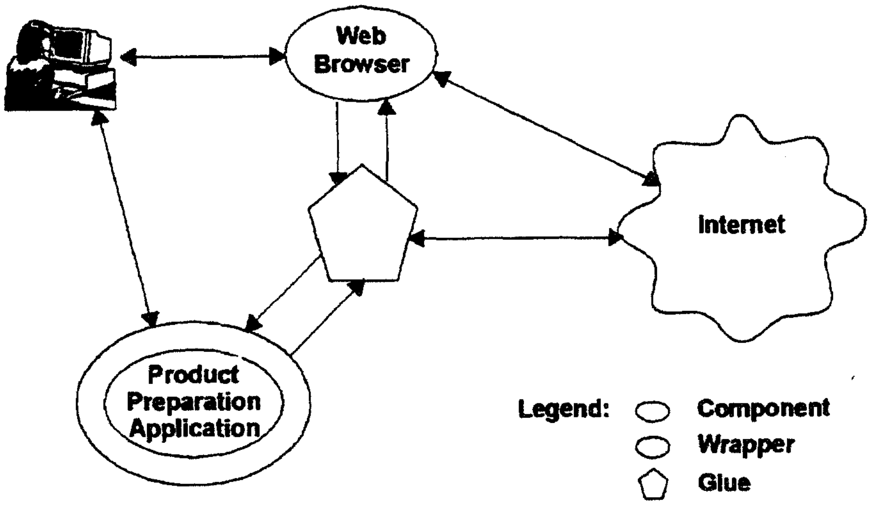


Figure 3.
Client side architecture.

preparation application so that it has a standard and known interface. By writing wrappers for applications from different vendors the system can be configured with different processing applications based on the preferences of the user. Subsequent substitution of any component involves modification to the wrappers only and not to the COTS component nor to the glue.

4. Strategies for Integration

The purposes of a software architecture are to define the major components of a system, how the components interface with each other, and the interactions between components to provide the system services. In a COTS based development approach many of the components will be commercial components. This puts some immediate constraints on the architecture:

- The architecture must adapt to the connectors available in the components being used. For older and legacy components this could involve such primitive techniques as screen-scraping and terminal emulation; more modern components may export interfaces available through ActiveX, JavaBeans, or CORBA.
- The architecture must adapt to the functionality available in the components. This includes adding functionality that is desired but not included in the component, as well as hiding functionality that is included but the designer wishes to mask out.
- The architecture must adapt to the possible different data models and data formats used by the diverse components. This involves a process of data mapping or translation to ensure appropriate communication channels between COTS components

Components as received from the component supplier do not, in general, conform to the architectural requirements of the integrator. The functionality available in the component does not correspond to the functionality the integrator wishes to see in the component; and the interface to the component is non-standard or does not conform to what the integrator wishes. In order to minimize the architectural mismatch found in the component the integrator must adapt the component to the desired architecture while simultaneously adapting the architecture to the COTS components available.

While the majority of the systems components will be COTS-based, there will inevitably be a need to design custom components that interact with the COTS. These components provide added functionality for which a suitable commercial application cannot be found. The design of these components can follow traditional design concepts but, in order to ensure a consistent architectural approach, these custom applications should be built so that they can be integrated into the system in a manner consistent with the integration of the COTS components. This includes the use of wrappers and glue, and consistency with the architectural style of the COTS components.

In order to design a system primarily for integration, we have identified a number of software integration components. These integration components serve different purposes and each has a different relationship to the components being integrated. The integration components that we use are: wrappers; glue; and tailoring. These are described in the following sections.

4.1 Wrappers

A wrapper is piece of code that the integrator builds to isolate the underlying COTS component from other components of the system. There are a number of reasons why a system architecture should include wrappers around components:

- Conform to standards, e.g., CORBA wrappers around legacy systems [1,7,9,16].
- Reduce the impact to the system of changes to the wrapped component.
- Provide a standard interface to a range of components. For mature domains a wrapper may be provided by the component supplier, e.g., ODBC. Standard interfaces are a prerequisite for substituting COTS components from different vendors.
- Add (or hide) functionality of a component.
- Give system integrator control over the "look and feel" of the component. Even though the integrator has no control over the component, the integrator does have access to the source and control over the wrapper.
- Provide a single point of access to the component.

One example of wrappers in the distributed imaging system is the ODBC interface to the Microsoft Access database. ODBC provides an industry standard API for accessing relational databases. By restricting access to the database to be ODBC compliant we can configure a system with a different database product by maintaining a similar data schema.

Another example of a wrapper is the script around the product development application. This wrapper provides a standard for moving data into the application. Unlike ODBC, which is an industry standard, this interface will be controlled by the designer. It will allow systems to be configured with different product development applications and for new applications to be plugged in as they become available.

4.2 Glue

The glue code provides the functionality to combine the different components. Purposes of the glue include:

- Control flow. Invokes functionality of the underlying components as required.
- Component bridge. Glue code can resolve any interface incompatibilities between components, for example by performing any necessary data conversions.
- Exception handling. By trapping exceptions the glue code can provide a consistent exception handling mechanism.

Within the distributed imagery management system, the server side glue code has been developed in Perl; on the client it is being developed in Visual Basic. In both cases they serve primarily as middleware combining components to add system functionality. For example, on the server there is a script that receives an image and cataloguing information from a client, invokes a component that converts the image format and creates an image thumbnail, stores the thumbnail and image, and updates the database. In the current configuration the database is Microsoft Access made available through ODBC; the image manipulation software is an ActiveX component with a Visual Basic wrapper to provide a standard interface; and operating system calls are used for file access and manipulation. The glue code has been developed in Perl and, with the exception of the ODBC which is specific to Win32, is platform independent. Use of JDBC is planned in order to make the code truly platform independent.

An example of a glue component on the client side is the middleware between the local browser, remote database, and local product preparation application. This component performs the following actions: inputs the found set and downloads the images in the set; opens a new project in the product preparation application; moves the images into this project; opens the GUI to the product preparation application on the users desktop. This glue is being written as a Visual Basic component.

4.3 Component tailoring

Component tailoring refers to the ability for system integrators to enhance the functionality of a component in ways that are supported by the component vendor. The tailoring is done by adding some element to the component to provide it with functionality not provided by the vendor. Tailoring does not involve modifying source code of the component.

Examples of tailoring include "scripting", where an application can be enhanced by executing a script upon the occurrence of some event. Early versions of scripting include simple macro languages. The scripting capability in many newer applications has become more sophisticated with full-fledged programming languages and interpreters, such as VBA, tcl and Perl, being accessible within applications.

Another example of a tailoring capability is the use of plug-ins. A plug-in is a component that registers with the enclosing application. The enclosing application makes a call-back to the plug-in when its functionality is required. By publishing the registration and call-back techniques, COTS vendors provide COTS users with a method of enhancing the component functionality without access to the source code.

When tailoring components in this way, designers must remember that the tailoring aspects are components in their own right. The designer must treat them as separate configuration items, make sure they are installed with the corresponding container component, and make sure that they are carried along during the upgrades.

Although we do not currently use tailoring in the distributed imaging system, one potential use is with plug-ins for the web browser. Browsers can display only a limited number of image types (typically GIF and JPEG) and do not provide editing capability for these images. By using plug-ins for Netscape or Microsoft browsers we can enhance their functionality to display more types of image formats and allow users to markup and annotate the images from within the browsers.

5. Rules for integration

Having developed a number of systems that use off-the-shelf components we have begun to develop a set of rules-of-thumb for easing the task of COTS component integration. Through experience we have found that complying with these rules simplifies the task of development and evolution of systems. These rules are outlined and discussed in the following sections.

5.1 Wrap all components

Rule number one states that all off-the-shelf components should have wrappers placed around them. Many of the following rules depend on such wrappers being in place.

The justification for wrapping all components is that the wrapper provides the only mechanism by which the integrator can control the interface and interactions of the component, and isolate other components of the system from changes to the underlying off-the-shelf component.

In addition, any custom software which is used to provide significant functionality in the system should be treated in a manner similar to a COTS component and integrated into the system using a wrapper. This will allow us to more easily substitute a COTS component for the custom component should future commercial developments provide the required functionality.

There are numerous examples in our work of where we have wrapped components. One example, on the server side of the image processing system, is the image manipulation component. This component is used for simple image manipulation functions, primarily format and size conversions. When looking for COTS components to provide this functionality we noted the following points:

- There are a wide range of possible solutions, each with its own interface. Even those within a single technology standard (e.g., ActiveX) had widely divergent interfaces.
- The functionality available in most potential COTS components far exceeded the functionality we required from the component. Masking out the unneeded functionality is therefore a significant concern.

By wrapping the component in a Perl module and exporting the appropriate objects we exported only the functionality of the component that we wished other programmers to use. It also allowed us to control and standardize the interface so that a substitution could be performed on the underlying component with no impact on other components of the system.

A different example of wrapping can be found in the client side of the imaging system. Certain clients contain specialized applications for product preparation and manipulating images. Users at these client stations should be able to find and download images from the database(s) directly into a project accessible by this application. We do not wish to state a priori what this image manipulation application is, and indeed want to support many such programs and leave it to the user to determine his/her favorite. Since there is no standard interface to these applications we needed to define and control the interface. The interface we have defined is primarily used to open a project and move the images from the network into the project. Once the project is established the image manipulation application is invoked and the user deals with the (familiar) interface of the application. In this case the wrapper is responsible only for the invocation of the COTS application and the establishment of the project files.

5.2 Make glue independent of underlying components

The glue provides functionality such as data and control flow, exception handling, and data conversion. Glue should be independent of the underlying components and should not change as different underlying components are substituted. Component changes should be hidden by the component wrapper and tailoring. Glue code only interacts with COTS components indirectly through the wrapper.

The functionality provided by the glue code should not depend on the specific off-the-shelf component that is being accessed. Services such as exception handling and control flow should evolve independently of the underlying components being glued together. As the components evolve and are modified, the glue need not be changed. By providing insulating wrappers around the components the glue can use standard methods for accessing the components.

In the image system, there are a number of examples of glue that is independent of the components. In the server system, the glue middleware is responsible for receiving images over the internet, using the image processing component, and inserting the image and cataloguing information into the database. By having standard interfaces to the imaging component and the database different components with similar functionality can be substituted by conforming to the standard interface without modifying the glue.

A similar example can be found on the client side for the glue that downloads images from the internet and places them in a project in the production application. A wrapper providing a standard interface to the production application allows for this glue to remain the same when a new production application is substituted.

5.3 Verify component version compatibility.

Components, particularly COTS components, evolve rapidly with frequent releases of new versions. Systems often have dependencies as to which versions of components will operate together. If a particular component is upgraded to a new version, this frequently means that wrappers and other components must be upgraded as well.

Designers and implementors should verify, whenever possible, that the current configuration of components is version compatible. Ideally this version checking is automated. The verification can be done either at build time, if all components are bound together at build, at installation time, or at run time if late binding is used between components.

Both Perl and Visual Basic using ActiveX components have some support for version verification. With Perl, this is done by the component developer including a version number with each released module. The component user specifies a required version number when linking to the module. A run time check is performed to verify that the module being linked has a version number equal to or higher than the required version.

ActiveX components have both a component version and an interface vérsion for each component. The component version is allocated by the component developer. It is used by the installation utility to determine whether a currently installed version of a component should be replaced with a newer version.

The interface version of a component is generated automatically when the component developer compiles the component. It records the compatibility between the interfaces of different versions of a component. If a component developer is creating a new version of a component with an interface incompatible with the previous version, a warning is given. At run time, when a Visual Basic program links to the ActiveX component, a check is automatically performed to verify that the interface exported by the component is the one expected by the Visual Basic program. The integration programmer can trap this exception if incompatible versions are installed.

5.4 Add assertions to the wrappers/glue

We are finding it useful in many cases to provide a high level of checking within the wrappers and glue in order to verify run-time assertions. The assertions can be as simple as verifying parameter types or values, or more complex, asserting relations between data values or required temporal orderings of events. Since the components are black-box, the glue and wrappers are the primary means by which developers can perform this type of run-time checking. By placing assertions between the components and raising an exception if they are violated, faults can be quickly detected and isolated within the system.

A simple example in the client software is the use of an assertion in the product preparation application wrapper that verifies the data type of the image being passed can be imported by the application. If a server provides image data in a format not recognized by the application (and we do not know a priori all the image formats the servers will provide nor all the product preparation applications that clients will use) an exception can be raised immediately and the user notified of the problem.

5.5 Do not have components talk directly to each other

Off-the-shelf components should always have some wrappers and glue between them and not interact directly with each other. This allows some tolerances in how precisely the components must fit together and minimizes the coupling between components. As a components evolves, the wrappers and glue around the component can be updated but there is minimal impact on other components of the system.

A second reason for placing integration components between all COTS components is that this integration code is the only source code to which the developer has access. If there is any requirement for developers to add to the system extra capabilities for testing, debugging, fault isolation, or instrumentation, it must be done inside the glue and the wrappers[13]. Even in the case where a particular COTS component could interact directly with others in the current system, future versions may exhibit different characteristics.

5.6 Be compatible with open standards

Selecting COTS components that depend on proprietary standards can cause problems with portability of the systems and with configuring a system with components from different vendors. With closed standards developers become locked into a specific vendors product and cannot move to another vendors product even if there are other products with similar functionality. As the different software domains mature, more and more standards are being evolved in the different application domains, and market pressure often forces vendors to comply with these standards. Pulling in the opposite direction, vendors develop proprietary and closed standards with added functionality to give them a competitive advantage and lock in customers to their product.

Within the imaging system we have tried to conform to open standards wherever possible and avoid closed proprietary standards. This assists in providing plug-and-play of components from different sources, and in configuring the system for different requirements. For example, we adhere to the standards specified in HTTP, HTML and CGI. This allows us to easily configure the system for any Web browser and server.

5.7 Avoid early commitment to an architecture

If system integrators wish to take advantage of available COTS components, many of the architectural and design decisions must be taken concurrently with the component selection process. By committing to a specific architecture and specific technologies too early in the process the system developers may make the integration of components difficult or preclude the use of COTS components entirely.

6. Conclusions

COTS software components have been used by software developers for many years, but with the increased complexity of software systems and the new and emerging technologies this trend is increasing dramatically. Although COTS software integration is being done, the approach has tended to be ad hoc resulting in systems that are error prone and difficult to maintain. By carefully defining the architecture and design of the system many of the problems and issues raised by the use of COTS software components can be addressed within a more formal and defined process resulting in more reliable software that can evolve over time. This paper has presented the elements of an architecture for integration and defined some informal rules that facilitate this integration.

Acknowledgments

This research was supported by the Chief, Research and Development (CRAD) of the Department of National Defence.

About the Authors

Dr. Mark Vigder is a Research Officer with the National Research Council. He has a Ph.D. in Computer and System Engineering from Carleton University, Ottawa, and has twenty years experience with software engineering as a practitioner, researcher, and educator.

John C. Dean, CD is a Research Officer with the National Research Council. He has a Master of Mathematics Degree (Computer Science) from the University of Waterloo, and has extensive experience in software engineering, project management and education.

References

[1] R.C. Aronica and D.E. Rimel Jr. Wrapper Your Legacy Systems. *Datamation*, 42(12):83-88, June 1996.

[2] T. Bao and E. Horowitz. Integrating Through User Interface: A Flexible Integration Framework for Third-Party Software. In *Proceedings: The Twentieth Annual International Computer Software And Applications Conference*, pages 336-342, IEEE Computer Society, Aug. 1996.

[3] A.W. Brown and K.C. Wallnau. Engineering Of Component-Based Systems. In *Proceedings of the 1996 2nd IEEE International Conference on Engineering of Complex Computer Systems*, pages 414-422, 1996.

[4] O.A. Bukhres and J. Chen and W. Du and A.K. Elmagarmid. *InterBase: An Execution Environment for Hetrogeneous Software Systems.* IEEE Computer. 26(8)57-69, Aug. 1993.

[5] J.C. Dean and M.R. Vigder. System Implementation Using Off-the-shelf Software. In *Proceedings of the 9th Annual Software Technology Conference*. Department of Defense, 1997.

[6] D. Garlan and A. Robert and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *17th International Conference on Software Engineering*, pages 179-185, 1995.

[7] D.L. Moniz. Integrating Legacy Databases Into a Common Infrastructure Using CORBA. In *Proceedings of the 9th Annual Software Technology Conference.* Department of Defense, 1997.

[8] J.K. Ousterhout. *Scripting: Higher Level Programming for the 21st Century*. Unpublished manuscript, available at http://www.sunlabs.com/people/john.ousterhout/scripting.html.

[9] C.M. Pancerella and R.A. Whiteside. Using CORBA to integrate manufacturing cells to a virtual enterprise. In *Proceedings of the SPIE Volume 2913*, pages 148-173, The International Society for Optical Engineering, 1997.

[10] M.Shaw and D.Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall Publishing, 1996.

[11] K.J. Sullivan and J.C. Knight. Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse. In *18th International Conference on Software Engineering*, pages 220-229, Berlin, 1996.

[12] M.R. Vigder and W.M. Gentleman and J.C. Dean. *COTS Software Integration: State of the Art*. National Research Council of Canada, Institute for Information Technology technical report NRC39198, January, 1996.

[13] J. Voas and G. McGraw and A. Gosh. Gluing Together Software Components: How Good is Your Glue. *In 14th Annual Pacific Northwest Software Quality Conference*, pages 338-349, Oct, 1996.

[14] L. Wall and T. Christianson and R.L. Schwartz. *Programming Perl 2nd ed.* O'Reilly & Associates, Inc. 1996.

[15] *Microsoft Visual Basic Programmer's Guide.* Microsoft Corporation, 1995.

[16] Common Object Request Broker: Architecture and Specification. Object Management Group.

[17] *USENIX Very High Level Languages Symposium Proceedings.* USENIX Association, Berkely, California, 1994.

Institute for Information Technology   Institut de technologie de l'information

NRC·CNRC

Building Long-Lived Systems from COTS Software Components

Dr. Mark Vigder
John C. Dean, CD
Institute for Information Technology
National Research Council
Ottawa, Canada
{mark.vigder|john.dean}@nrc.ca

Canadä

Software Engineering Group

Staff

- Nine researchers and two support staff

Facilities

- Access to most of the modern development environments
- Computer Zoo
  - various hardware platforms

NRC·CNRC

Group Interests

- Software reuse
- Configuration management
- Commercial off-the-shelf (COTS) Software
- Real-time and embedded systems
- Formal methods in software engineering
- Software architecture
- Human Computer Interaction in software engineering
- Consortium for Software Engineering Research (CSER)

COTS Project

- Explore issues associated with using COTS software components to build long-lived systems
- Taken from the perspective of a system integrator
  - integrates COTS components
  - does not develop commercial components

Goals

- provide guidance for procurement, development and administration

What is a COTS component?

A software component that has been bought from a third-party and that the developer uses on an as-is basis.

- User of the COTS component does not modify the source in any way.
- COTS developer is responsible for maintenance and evolution of the COTS component.
- Identical copies of the COTS component have been sold and are being used by different developers.

Examples of COTS components

- Subroutine, Abstract Data Type, or Class
- Library
- Generic service, eg. database or GIS
- Complete application
- Application generator
- Framework accepting plug-ins and specialization

Viewpoints to consider

- users
  - work with complementary COTS products from various suppliers
- system integrator
  - builds custom system from COTS components
- COTS product developers
  - incorporate other COTS components to create a new COTS product

Attractions of COTS are compelling

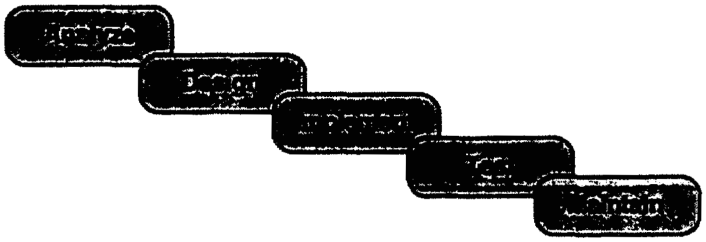
- Better time-to-delivery
- Lower amortized development cost
- Lower amortized support cost
- Better quality through maturity
- Better concept of product
- Access to scarce expertise
- Reduced user training

COTS has risks

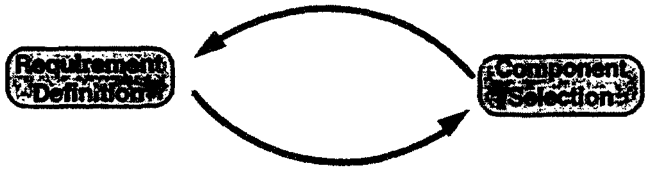
- Fitting the application to COTS may be challenging
- Detailed specifications often unavailable
- Issues of concern may be unknown
- Needed functionality may be missing
- Undesirable functionality may need to be masked
- Unanticipated limitations and faults may turn up
- Maintenance and evolution is not yours to control

Impact of using COTS

- **On the process**
- On architecture and design
- On maintenance



Constraints on Process



- Requirements gathering and component selection are concurrent activities
- Component selection impacts architecture of system.

Revised development process

- Different requirements analysis and specification
- New step of component selection (involves customer)
- New step of component testing and modelling
- Changed high level design; less detail design
- Less coding and unit testing
- Changed integration testing
- Delivery
- Changed post-delivery support

Selection among COTS Alternatives

Resource intensive yet still uncertain

- Choice involves more than just technical issues
- Typically difficult to find sufficient information
- Restricted access to proprietary information
- List of pre-qualified components doesn't work
- Would access to real source code help?

Component Assessment

Assess component characteristics:

- implements protocol X, vendor support...

Assess within context of the application:

- Criteria (functional, performance, architecture, lifecycle, look-and-feel...)

Techniques for assessment:

- generic testing, user community, vendor literature...

Risk assessment and mitigation:

- Vendor drops product, does not meet requirements, bugs in product,...

Evaluating and Modeling Components

Evaluation in the context of intended usage

- Quantitative assessments of performance
- Quantitative assessment of resource consumption
- Quantitative assessment of usability
- Quantitative assessment of reliability

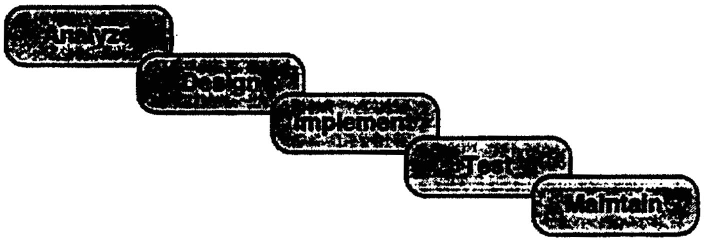
Evaluating and Modeling Components

Evaluation in the context of intended usage

- narrowly focused
  - specific capabilities
  - detailed tests
- limit candidates
  - reduces evaluation time
- not competitive

Impact of using COTS

- On the process
- **On architecture and design**
- On maintenance



Constraints on system architecture

- Highly distributed.
- Black-box, shared, components with "glue" to hold them together.
- "Plug-and-play" architecture.
- Tailoring of pre-built components.
- Multiple specialized components
- Dynamically reconfigurable
- Services added and modified

Technology available

- Open standards (HTTP, LDAP, MIME, SNMP, O/JDBC, Java(?)...)
- Distributed objects (DCOM, CORBA, RMI)
- Scripting (languages, interfaces...)
- Component technology (Java Beans, ActiveX,...)
- Frameworks with plug-ins and inheritance
- Design patterns (facades, adapters, mediators)

Change in Design/Implementation

- Not a top down process
- Wrapper based architecture maps representations
- Glue coordinates control flow, c oncurrency, errors
- Tailoring of COTS components
- Masking unwanted functionality
- Missing components
- Always plan for component change

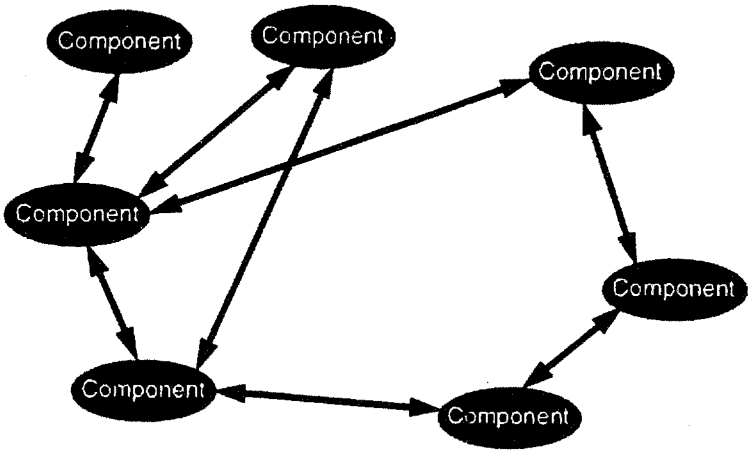
Checklist of items to validate

- correct use of design patterns
  - all components are protected by an *adapter* (wrapper)
  - components grouped into subsystem are protected by a *facade*
  - no direct component interaction; done through a *mediator*

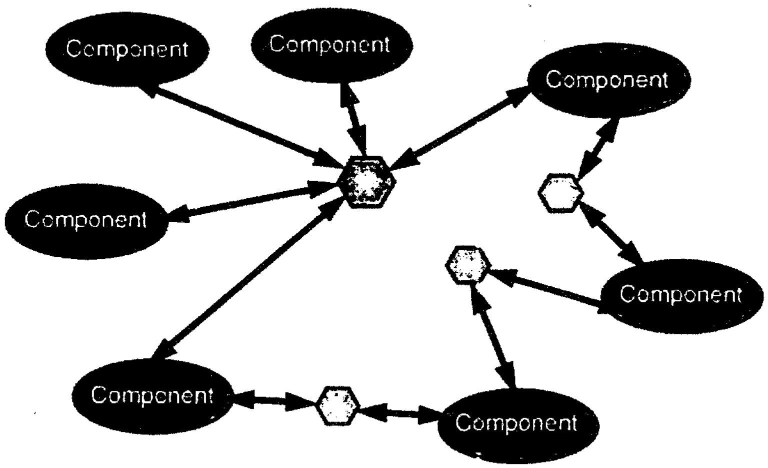
Checklist of items to validate...

- instrumentation code
  - identify static constraints and configurations
  - determine dynamic behaviour

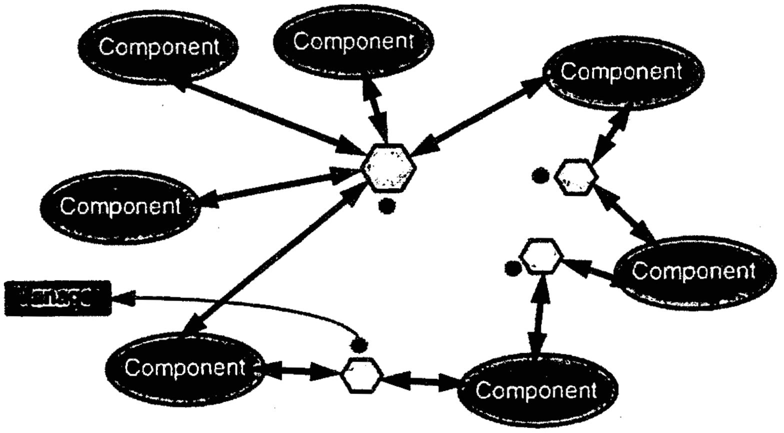
Random
Component Architecture



Organized
Component Architecture

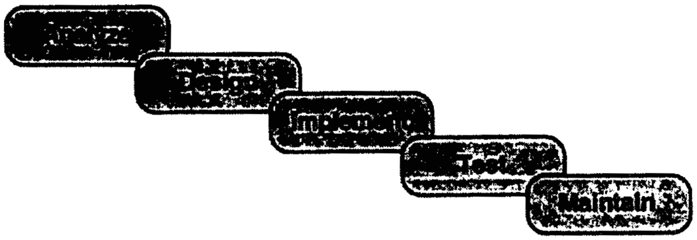


Instrumentation



Impact of using COTS

- On the process
- On architecture and design
- **On maintenance**



Change to ongoing maintenance

Component replacement.

- New versions of a component
- Similar component from different vendor
- Verifying compatibility of components

Component management

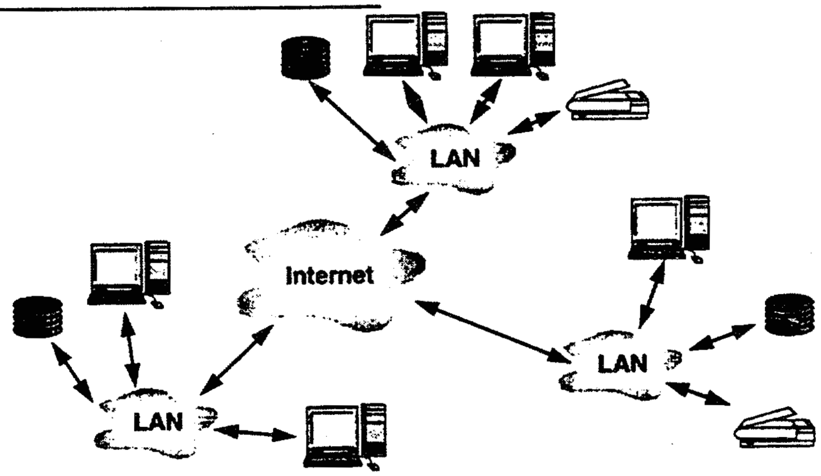
- Tracking COTS components
- What's installed where?
- run-time monitoring for fault isolation/identification; requires instrumentation and management hooks built into glue code

The IDTMS Prototype

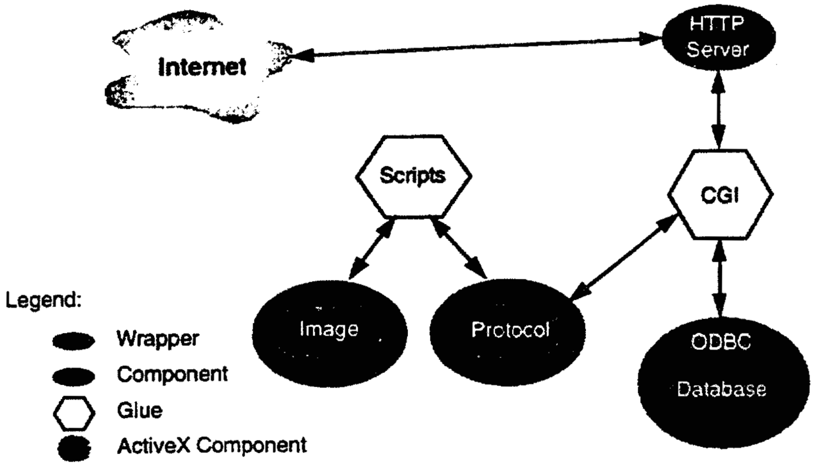
Image Document Transfer and Management System

- Work with Canadian Forces Photo Unit
- Web-based client-server system
- used to demonstrate concepts

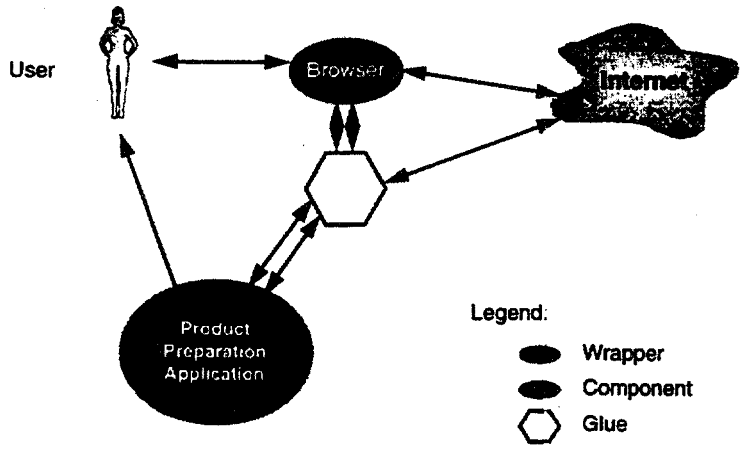
Distributed architecture



Server Architecture

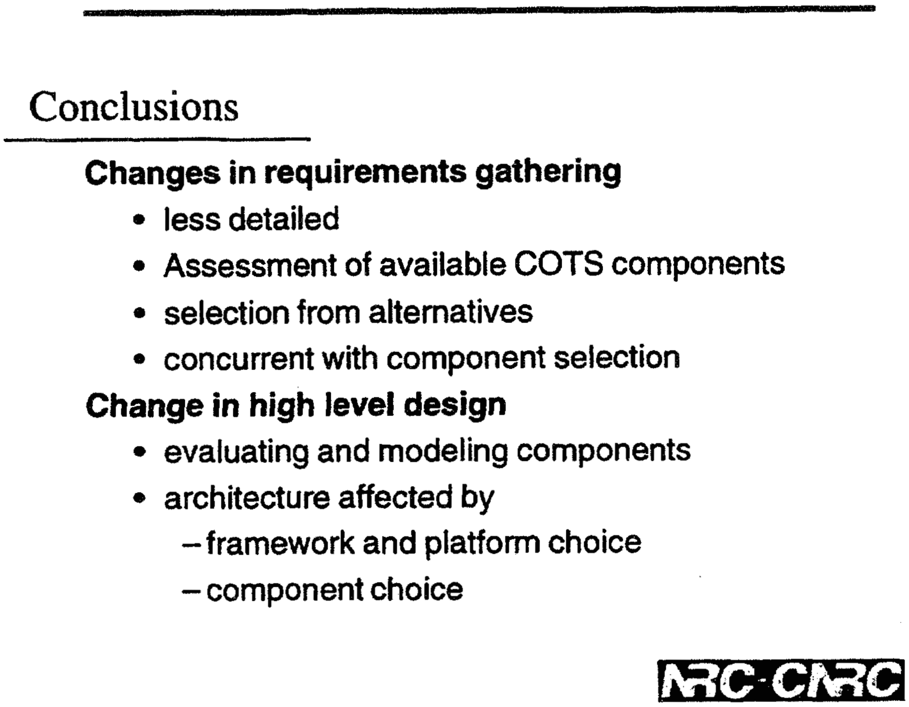


Client Architecture



Challenges

- Scaleable architecture
- Replaceable components
- Component management
- Re-use existing infrastructure
- Component integration
- Incremental development and deployment



Conclusions

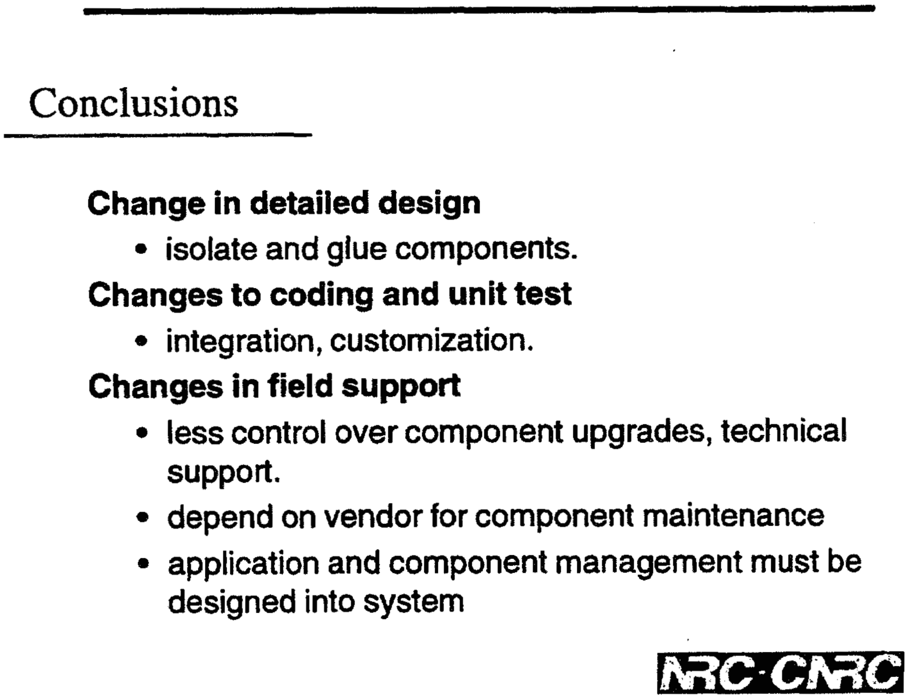
Changes in requirements gathering

- less detailed
- Assessment of available COTS components
- selection from alternatives
- concurrent with component selection

Change in high level design

- evaluating and modeling components
- architecture affected by
  - framework and platform choice
  - component choice

NRC·CNRC



Conclusions

Change in detailed design

- isolate and glue components.

Changes to coding and unit test

- integration, customization.

Changes in field support

- less control over component upgrades, technical support.
- depend on vendor for component maintenance
- application and component management must be designed into system

NRC·CNRC

A Software Development Process for COTS-based Information System Infrastructure

Greg Fox, Technical Fellow, TRW Systems Integration Group
Karen Lantner, Systems Engineering Manager, EDS
Steven Marcom, Senior Systems Analyst, TRW Government Information Systems

Abstract

Modern software developers are guided by a variety of formal and informal processes that organize and control development activities across large groups of developers or multiple organizations and supply discipline and order lacking in many early development efforts. The available inventory of documented process methods is limited: Most process methods assume the system being built will be coded largely from scratch. The processes do not address many of the challenges associated with building systems that contain large amounts of commercial off-the-shelf (COTS) software. The Infrastructure Incremental Development Approach (IIDA) is a combination of the classical development model and the spiral process model to accommodate the needs of COTS-based technical infrastructure development.

1. Background

The level of abstraction at which the software developer works has changed markedly throughout the last 40 years. Early programmers used ones and zeros to control the electronic switches within computers. That technology was followed by procedural languages that, from the programmer's view, removed much of the physical housekeeping associated with the specific design of the computer. In recent years, an even higher level of abstraction has appeared: the integration of prepackaged commercial off-the-shelf (COTS) software into system designs. In addition, the domain of software development has become segmented into different layers. For example, application-level software development can be distinguished from infrastructure-level software development.

1.1. The Emerging Divide in System Functionality

The value of layering in software architecture and implementation is an established concept. Key to the layering model is the idea that through use of defined interfaces between layers, the impact of changes in any given layer can be largely isolated from the other layers.

The concept of a services layer and of specialized software in the system acting as service providers has continued to grow from the simple beginnings in the operating system to become a fundamental architectural concept in modern system design. As reuse and portability of software applications across different vendor hardware platforms become an increasingly important goal, a more sophisticated model of service layers and service providers has emerged. The open systems movement cites application portability across computing platforms as a major economic driver. [1] [2]

The National Institute of Standards and Technology (NIST) Application Portability Profile (APP) [3] provides one convenient model for defining the system layers and services that support portability. This model, along with standards, can be used to achieve application portability by guiding designers who plan to code new information systems in their entirety and by guiding selection of available software computing components from those available in the marketplace.

Modern information system design models separate the business-specific application software layer in a system from the technology-based infrastructure software layer. An illustration of this approach is the information engineering method of separating business system architecture from technical architecture, which contains the computing infrastructure. [4] This separation into software layers, which is less formally addressed in other design methodologies, recognizes that change and evolution in information systems are driven by two independent forces: change in business requirements and change in technology. Decoupling the impact of business rule change from change in technology decreases the total amount of system rework necessary to support system evolvability over time. This decoupling is effectively implemented by modeling the infrastructure software using the concept of services layers and service providers.

0-8186-7940-9/97 $10.00 © 1997 IEEE

1.2. Views of Infrastructure

There are two ways of looking at the infrastructure. One view is the services view of infrastructure as seen by business application developers. It includes Human Computer Interface, Systems Management, Security, Workflow Management, Telecommunications, Data Interchange, Transaction Processing, Data Management, and Operating Systems. This grouping of infrastructure services was derived from the NIST APP. Infrastructure services are delivered to the applications through an application programming interface (API).

The second view is the structural view, which includes the kinds of components infrastructure developers use to construct their view of the infrastructure: a set of connected software, network, and hardware components. These include developed software components, COTS software components, communications circuits, local area networks, special purpose servers, general purpose servers, workstations, and laptops.

An additional set of functionality treated as part of the infrastructure during the development process are the technical applications needed to operate the system. These applications neither implement business functionality nor provide services to the business application. They are, for example, the tools for system security administration, database administration, system configuration control, and software distribution and, in general, the toolset for enterprise-level systems management. Other infrastructure services are also used internal to the infrastructure but are not visible to business applications or end users. For example, a remote data access protocol is a level of service provided between infrastructure components that is used to construct a mechanism for accessing data: It is not directly visible to business applications or end users.

The infrastructure services provide functionality that the application developer can access external to the application and, therefore, does not develop as part of the application. Economy of scale is achieved through the common use of technical services by application development projects across the enterprise. Programmers can access infrastructure services without regard to how the underlying infrastructure services have been implemented using a properly designed API. By allowing application and infrastructure development to be separate and independent, infrastructure enhancements (e.g., increased performance, additional services, and new computing platforms) can be made with minimal effects on application development.

1.3. The COTS Challenge for Infrastructure

Although distributed systems (popularly described as client/server or networked systems) dominate today's computer system design, they still have the character of adolescence. We are in the middle of a dramatic, and somewhat uncontrolled, expansion and evolution of standards for COTS software products for distributed systems. COTS products provide portions of the needed supporting technical functionality to turn collections of computing platforms into unified, distributed computing environments. The available COTS software products offer varying degrees of standards compliance, interoperability, heterogeneous computing platform support, security functionality, performance efficiency, and distributed environment transparency for applications using their services.

Two separate panels at the 1995 SEI/MCC Symposium on the Use of COTS in Systems Integration concluded that "... there is a need for process definitions for COTS usage" [5] and "... new life cycle models for COTS integration projects are needed." [6] Currently, documented software development life cycle processes provide little practical guidance to developers to achieve the advantages of COTS software or to assist in the selection of specific products from the myriad available. COTS product selection and integration are complicated by an intrinsic set of special characteristics: incompatibility, inflexibility, complexity, and transience.

1.4. Development Life Cycle Process Impact

The special characteristics of COTS software integration change the emphasis in the classic waterfall life cycle stages of planning, definition, analysis, design, construction, integration/test, implementation /deployment, and maintenance. COTS-based development differs from business-application-oriented development in that the COTS selection process must occur early in the life cycle COTS evaluation and selection become a critical part of the early analysis process rather than a peripheral activity within the later design process. The challenges of COTS incompatibility, inflexibility, complexity, and transience must be addressed in the selection process because the infrastructure will ultimately consist of a suite of COTS products that must operate in harmony.

In addition, since COTS software does not require coding but does require integration with other components, it starts the life cycle as a partially developed component. The design, construction, and integration/test development stages must be recast to accommodate early COTS software integration and testing as well as to develop glue code: interface software, configuration files, scripts, utilities, and data files required to make the COTS software deliver its intended functionality. The proper development and testing of the glue code to make a COTS package work may not be a trivial undertaking. For more complex COTS software, the development of glue code might need to be treated in the same manner as the development of a traditional custom-coded software module.

When a COTS product enters the development process, the first task is to test and integrate it into the system. This activity starts early in the development process. Waiting until late in the development process to test and integrate COTS products, particularly complex ones, will not give adequate time to master all their intricacies and complexities. COTS product testing and integration activities must be interwoven into more of the development process stages.

1.5. COTS Incompatibility

Many vendors do not develop their products along the lines of the layering models discussed earlier. At this time, no single commercially available software product or product family can provide all the infrastructure services needed for an enterprise-level information system of substantial size or complexity. The problem to be solved in system design and development is to select a compatible set of software products that can be integrated together and augmented by glue code to produce a complete set of services.

In an ideal world, a set of products that provide all the needed infrastructure services would simply "snap together" like the pieces in the puzzle shown in Figure 1. In the real world, this is not the case: When put together, the COTS pieces have gaps and overlaps. At any point in time, the set of services that a system designer can specify as useful exceeds what is available in mature products in the marketplace. The resulting gaps can be overcome in two ways. One is by traditional design and development of custom infrastructure software added around the commercially available products selected (either adding layers between the COTS-based infrastructure and the applications or adding custom service-provider software that is conceptually parallel to the COTS software). Another way is by leaving it to the application designers to deal with at the application level.

Overlaps between products can cause a greater system design problem than gaps. Commercial software suppliers are driven much more by a desire to capture larger segments of the marketplace than they are by adherence to recommended system implementation layering models. For example, boundaries between database access, transaction processing, and workflow management software products begin to overlap and blur as each vendor community expands its product's features in pursuit of increased market share. This expansion in features is driven by requests for increased functionality by the installed base, not by the boundaries defined in layering models. The net result is that certain products and product sets do not work in a synergistic fashion with other products; yet none of the products on its own is complete enough to provide all of the necessary functionality. Selection of a specific product that provides a certain set of services often precludes selection of another functionally complementary product.

1.6. COTS Inflexibility

The inflexibility characteristic of COTS software can cause both design and integration difficulties. Unlike custom-developed software, when a piece of commercial software exhibits a behavior not expected by the system designer, the developer cannot simply change the behavior of that software but must either replace the software, work around the unexpected behavior, or change requirements. Understanding the behavior of an unmodifiable software component is a different process than specifying the behavior of a component to be constructed. Most documented software development methodologies take the latter approach and do not address the former.

1.7. COTS Complexity

The complexity characteristic of many of today's advanced COTS software products causes distortions in

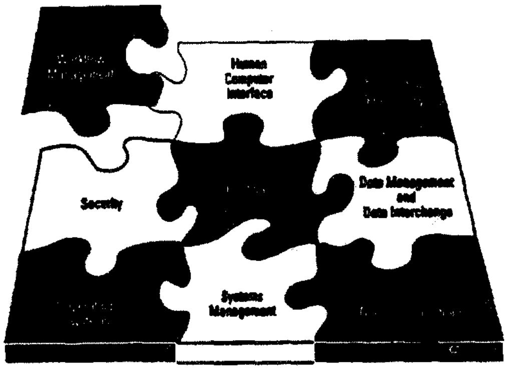


Figure 1.
Generic Information System Infrastructure Services

the traditional development process time line. The flexibility and tailorability of product families like transaction monitors, workflow managers, and system management frameworks mean a significant education investment. The investment must be made up front before the product can be fully evaluated for selection, and in cases when the product proves unsuitable, the investment might have a net zero return. Experience shows that the selection process for one major product can require 3–6 months of calendar time, multiple engineers and programmers, and access to sophisticated suites of hardware and software environments and will likely entail the purchase of vendor-provided training classes.

The more complex COTS software products are tailorable and scalable to multiple hardware configurations, software environments, and workload environments. To achieve this flexibility, they contain from dozens to hundreds of adjustable parameters (or "knobs"). Each of these must be set for the specific system configuration. If the system is being built for deployment in multiple locations with different hardware configurations or workload environments, the COTS software parameters might need to be tuned for each installation. This can be a complex task requiring product expertise, experience with the behavior of the integrated system, and, potentially, support from analytic modeling efforts. Software configuration files for each location might need to be tailored using the information developed during system integration. Not only does this require additional development effort, but the scheduling process must recognize that just because the system has been integrated and tested in a test facility does not mean that it can be quickly made operational at a production location. The tailoring and tuning process for each location's configuration can require days, weeks, or months to accomplish.

1.8. COTS Transience

COTS software products are characterized by periodic updates. Updates might add functionality but are often incompatible with other system components. On the other hand, remaining with older versions of COTS products might cause future interoperability problems with upgrades to other COTS software. COTS software updates, particularly operating system updates, must always be evaluated for insertion into the system since critical vendor maintenance and support for older versions often ceases. Management, cost, and technical factors in the transition to new COTS software versions can be formidable, particularly in a system with dozens of interrelated products being upgraded by their vendors on different calendar cycles.

2. The Infrastructure Incremental Development Approach (IIDA)

The development of a COTS-based technical infrastructure demands an approach that is fundamentally different from traditional approaches used for business-oriented applications: one that is heavily prototype-oriented, emphasizes testing, and evolves through multiple iterations. The IIDA is a tailored life cycle that preserves the benefits of existing structured processes for software development while adapting to the particular characteristics of integrating COTS products. The IIDA is a combination of the classical waterfall development model [7] and the spiral development model [8], but the emphasis is on establishing compatibility and completeness rather than on component-level specifications.

2.1. Overview of IIDA

The IIDA is an iterative and incremental approach to infrastructure development where each version of the infrastructure is an increment that is integrated into the existing infrastructure baseline. Within each version, development proceeds in time-sequenced stages with iterative feedback to preceding stages. (See Figure 2.)

The target infrastructure is the long-term vision for the infrastructure. It is defined and subsequently refined during the Definition and Analysis Stage through a top-down process of analysis of the enterprise requirements, enterprise adopted standards, and the system architecture. The Technical S rategies component captures the high-level description of the complete system vision and defines how the infrastructure will operate. [9] The Services Identification component is built up over time and is influenced by technology trends, product assessments, and the anticipated needs of the business applications.

Stages of the development cycle are augmented with a series of structured prototypes for COTS product evaluation and i tegration. For each COTS family, the prototypes evolve from initial analysis prototypes for a make/buy decision to, first, a series of design prototypes for COTS product selection and detailed assessment and, finally, to a demonstration prototype that becomes part of the development test bed. The timing of the prototypes is aligned with the development stages and the stages are dependent on the products from their corresponding prototypes. This close coupling of prototyping and classic development stages characterizes the IIDA.

Each pass through the stages in Figure 2 yields an incremental version of the infrastructure that can be integrated with applications and deployed. After the

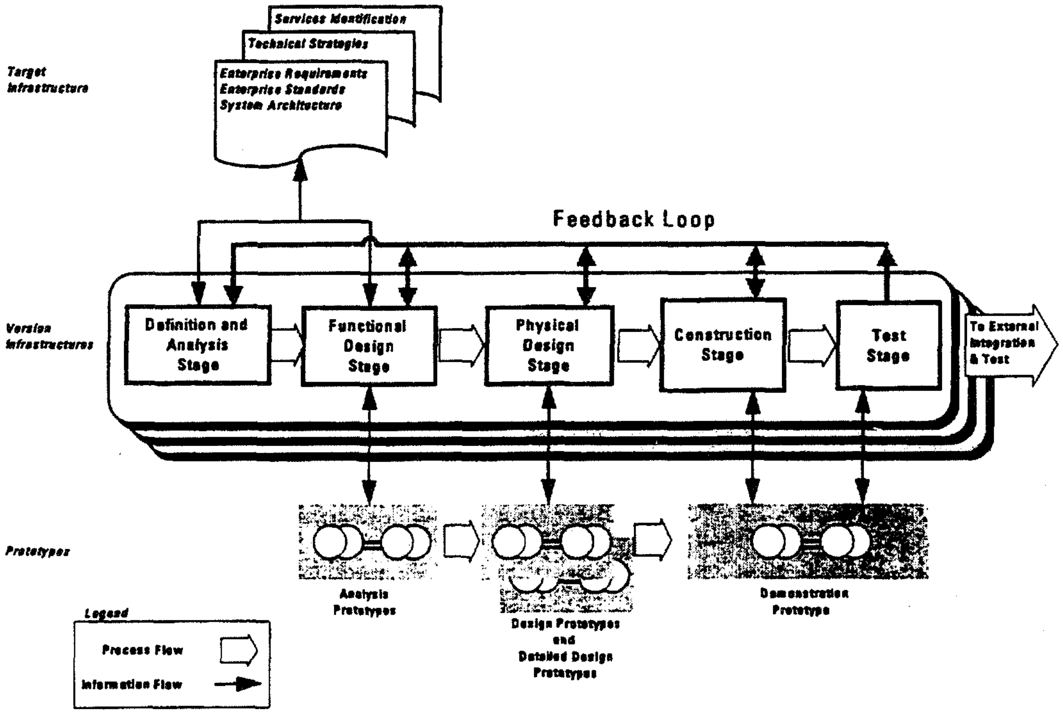


Figure 2. Infrastructure Development Approach

implementation of each version, successive developmental cycles are initiated. The infrastructure thus evolves towards the target infrastructure by providing an increased level of services to business applications and developers and by incorporating new underlying technology and products.

Infrastructure components are integrated into the existing infrastructure baseline. The components in this integrated infrastructure baseline are then ready to be integrated and tested with business applications. Infrastructure development ends with a technical platform upon which business applications can run effectively rather than with an operational product. The scope of this paper is infrastructure development: It does not include the external integration, testing, or distribution of business applications.

2.2. IIDA Stages

The following is a summary of the major activities of each IIDA stage:

- Definition and Analysis Stage
  - Enterprise requirements, enterprise standards, system architecture, and technical strategies are defined and refined.
  - Version-specific functional infrastructure requirements are established by considering business application areas, architectural imperatives, and technology availability.
- Functional Design Stage
  - Services included in the target and current versions are identified and defined.
  - Prototypes are used to identify leading candidate COTS components.
- Physical Design Stage
  - Interfaces between applications and infrastructure are defined. (API is established.)
  - Internal design of services is defined both functionally and technically.
  - COTS and to-be-built components are identified.
  - Prototypes are used to select and characterize COTS components.
  - Preliminary bill of materials (BOM) is created for acquisition of equipment and COTS software products.
  - Design is calibrated for scaling and performance considerations to provide site designers with site configuration guidelines.

- Structure of each to-be-built component and its interfaces is defined.
- Construction Stage
  - To-be-built components are constructed.
  - Glue code is developed and the unit is tested.
  - COTS components, glue code, and built components are integrated into the infrastructure using the demonstration prototype as a test bed.
- Test Stage
  - Infrastructure versions are tested prior to sending them to be integrated and tested with business applications.

2.3. IIDA Milestones and Deliverable Documentation

The infrastructure development approach uses both formal and informal reviews, turnovers, and walkthroughs to maintain the degree of formality necessary to control and communicate the design. (See Figure 3.) Formal reviews include:

- Technical Review at the end of the Analysis Stage
- Design Review during the Physical Design Stage
- Test Review at the end of the Test Stage.

Formal reviews are attended by organizations external to the infrastructure development group, as well as infrastructure developers and managers. These reviews occur once during the development cycle for each version of the infrastructure.

Other reviews, turnovers, and walkthroughs are informal, rolling peer, or management reviews that typically occur when pieces of the design, construction, or integration are ready to be walked through. Both infrastructure developers and managers participate in the following informal internal reviews:

- Top-level Design Walkthroughs during the Definition and Analysis Stage
- Design Turnovers (from design to development organization) during the Physical Design Stage
- Detailed Design Walkthroughs at the end of the Physical Design Stage
- Code Walkthroughs during the Construction Stage
- Test Design Walkthroughs during the Construction Stage
- Development Turnovers (from development to test organization) at the end of the Construction Stage.

The lower portion of Figure 3 shows the key documents that are produced during the IIDA process. Target infrastructure documents, which include Enterprise Requirements, Technical Strategies, and Services Identification are created once at the beginning of infrastructure development and updated as versions are produced. Version-specific infrastructure documents are created for each infrastructure version. Not shown in the tables are the informal documentation packages developed for the formal reviews and informal walkthroughs.

2.4. The Critical Role of Prototypes

At the heart of the IIDA approach is a series of tailored prototypes, shown as Analysis, Design, Detailed Design, and Demonstration prototypes in Figure 2, which also illustrates their respective time phasing in the overall process. This can be viewed as a tailoring of the spiral development model as each successive set of prototypes serves to narrow the solution space for the final implementation.

2.5. Analysis Prototypes

Analysis prototypes are used to identify leading candidate COTS software products in each COTS family. A COTS family is defined as a group of COTS software products that performs similar functions and/or provides related services to the application developers. Analysis prototypes are designed to exercise a COTS product to determine its general capabilities and to discover how well it satisfies the needs of the current version of the infrastructure. Selection of the best product in each family is performed later using the design prototypes. A sample application can be written to serve as a test vehicle for the family of products under evaluation because infrastructure, by its very nature, provides services rather than active applications. The results of the analysis prototypes feed the version-specific services definition and the version-specific services interface (API) efforts with information on available COTS product behavior and performance. A suite of COTS products will be recommended as a result of these prototypes that, when combined with custom-developed glue code and to-be-built software, cover all the requirements of the combined service areas.

Analysis prototypes are also used to examine emerging technologies for possible inclusion in future versions of the infrastructure. Technology insertion plays an important role in infrastructure evolution from version to version.

Through the analysis prototypes, methods to implement target technical strategies into future infrastructure versions can be postulated and developed. In this role, the analysis prototype is supporting the evolution of the definition of the long-term vision or target infrastructure.

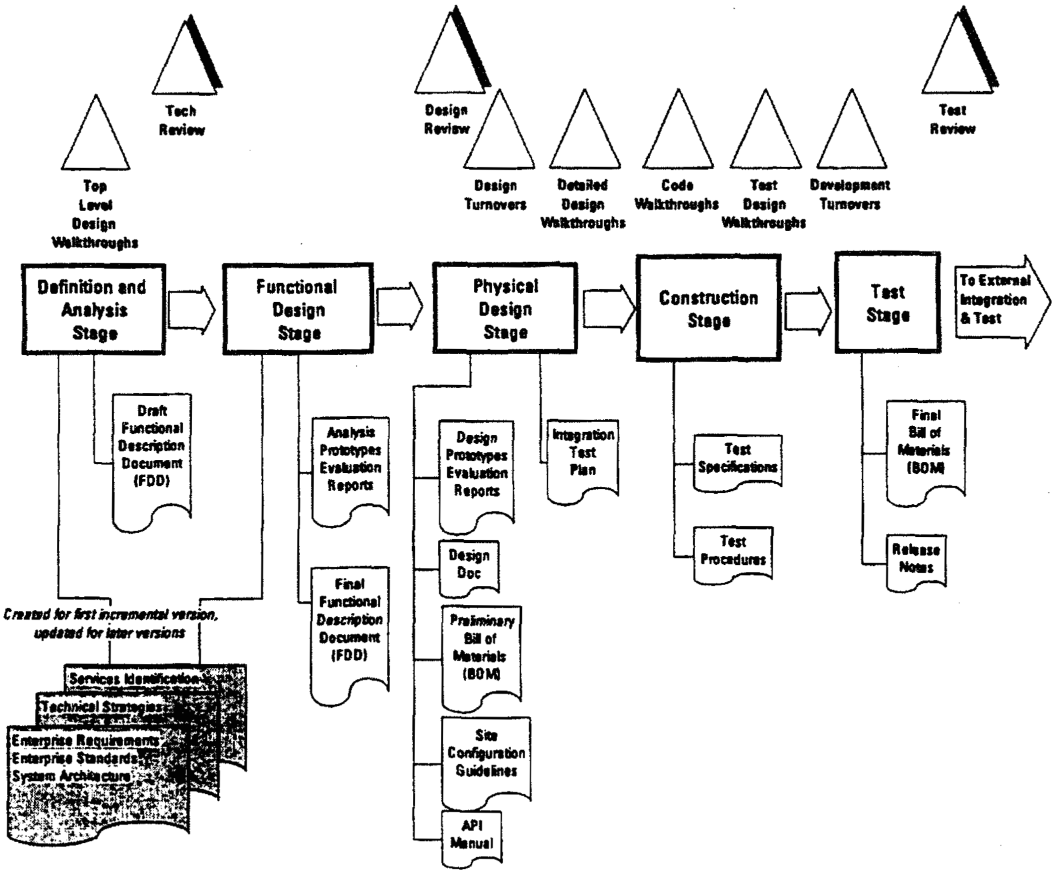

Tech Review
Design Review
Test Review
Top Level Design Walkthroughs
Design Turnovers
Detailed Design Walkthroughs
Code Walkthroughs
Test Design Walkthroughs
Development Turnovers
Definition and Analysis Stage
Functional Design Stage
Physical Design Stage
Construction Stage
Test Stage
To External Integration & Test
Draft Functional Description Document (FDD)
Analysis Prototypes Evaluation Reports
Final Functional Description Document (FDD)
Design Prototypes Evaluation Reports
Design Doc
Preliminary Bill of Materials (BOM)
Site Configuration Guidelines
API Manual
Integration Test Plan
Test Specifications
Test Procedures
Final Bill of Materials (BOM)
Release Notes
Created for first incremental version, updated for later versions
Services Identification
Technical Strategies
Enterprise Requirements
Enterprise Standards
System Architecture

Figure 3. Infrastructure Milestones and Deliverable Documents

2.6. Design Prototypes

The purpose of a design prototype is to select the best COTS product to incorporate into the design from several candidates in each area identified through the earlier analysis prototypes. A design prototype exercises a COTS product to determine its functional capabilities and how well it performs in accordance with its documentation. Specific benchmarks can be run in addition to functional tests. Sample applications will usually be written as test vehicles for the products under evaluation and to stimulate service performance under conditions that would be found in the application environment.

2.7. Detailed Design Prototypes

Detailed design prototypes are a special case of the design prototypes. They serve as proof-of-concept prototypes and are designed to exercise the selected COTS products to demonstrate that detailed COTS product capabilities are consistent with the design expectations. Sample applications are usually written to serve as a test vehicle for the products under evaluation. The results of the detailed design prototypes feed the services' detailed internal design with information on COTS behavior and performance and with specific language and syntax requirements for invoking services.

At this level of detail, designers might find that a COTS product does not perform as documented or as expected or that there are unexpected side effects of a product's behavior. This feedback is provided to the functional design activity, which might need to modify or redesign the solution with a substitute COTS product. The evaluation documentation created during earlier analysis and design prototypes is used to streamline alternate COTS selection.

2.8. Demonstration Prototype

The Demonstration prototype is used to unit test infrastructure components and to serve as a platform for infrastructure component-to-component integration. The sample applications used for the design prototypes might be reused if robust enough to exercise the elements tested in the unit test.

The results of the demonstration prototype feed back into the unit test activity. Unlike the analysis and design prototypes, which are investigative and throw-away in nature, the demonstration prototype is cumulative and evolves into a test-bed environment for the infrastructure.

3. Application of IIDA

The following text describes the experiences using the IIDA methodology from 1994 to 1997 to develop the initial versions of an infrastructure to support business applications developers for a large enterprise-wide heterogenous system. The types of COTS products that were integrated included:

- Operating systems provided by four different vendors
- End-user interface COTS software to provide a common graphical user interface (GUI)
- Middleware COTS products to provide a uniform transaction processing capability
- Combinations of COTS software and glue code for specialized services such as security and failover recovery
- Relational database management systems
- COTS applications for systems management, e.g., software distribution and remote database administration

3.1. Conventional and Unconventional Wisdom

Assumptions at the beginning of the development cycle were that the use of infrastructure COTS products would provide the following benefits:

- Using COTS products would reduce development costs and overall schedule.
- By corollary, the development cycle would be accelerated.
- Feasibility demonstrations could be put together quickly.
- End-product quality would be higher as measured by a richer feature set and increased system robustness (assuming the selected COTS product is mature). [10]
- COTS vendors would provide maintenance for their COTS products.

The experience integrating infrastructure COTS products and developed code refocused attention and revealed an additional set of assumptions for future developments:

- Accelerated development catapults you immediately into an integration and test activity.
- Hands-on evaluation requires early simulated applications in an integrated environment; these simulated applications and other test software can represent significant development costs.
- Maintenance on identified problems is provided by the COTS software vendor but problem investigation and identification by the integrator are the most costly parts of COTS software maintenance.
- Maintenance turnaround time by the vendors can be a significant problem.

3.2. Life Cycle Implications

The development methodology must be specifically tailored to accommodate COTS product integration. This entails a set of assumptions and constraints quite different from custom-built development.

The front-end processes in the definition and analysis stage must support concurrent requirements and COTS product analysis. The analysis prototype in the functional design stage must provide for iteration and a flexible linkage between the COTS product evaluations and the feedback loop to requirements analysis.

During the construction stage, the development processes acquire a dual nature when COTS product integration is introduced. One process path is valid for COTS product integration, and another process path is valid for developing the glue code and custom-built components. These two process paths are equivalent but consist of different activities and products. In addition, all COTS products, glue code, and custom-built components must be integrated together to complete development.

During the construction stage, the development of glue code that integrates COTS products and fills in missing functionality is similar to the development of traditional software, and the traditional process of coding, unit testing, and integration is applicable.

For COTS products, the construction stage is when COTS products undergo detailed tuning and configuration and when the interfaces and threads between components are exercised in a multi-COTS product environment. COTS product tuning, configuration, and integration have an analog to code and unit-test activities. Unit test with COTS products is black-box (versus white-box) testing, and the focus is on interfaces and COTS product behavior. For example, unit testing of the transaction processing monitor consisted of exercising all of the application programming interface (API) calls supported by the product as configured within the target environment.

The traditional software maintenance activities must be expanded in scope and extended to provide continuing COTS product support. This support starts early in the life cycle. Application developers must have early deliveries and training for partially completed infrastructure functionality to keep their development life cycle within reasonable time frames. They also require on-site, hands-on direct support from infrastructure developers and integrators to ensure acceptance and proper use of the infrastructure products.

Configuration control must be organized and in place early to accommodate multiple versions of the COTS products and configuration files. Separate environments for development and integration must be well-defined and structured to accept the delivered COTS products. Early support for multiple baselines must be in place as the combinations of COTS products become complex.

Throughout the life cycle, feedback loops allow ongoing reevaluation of the COTS products. Analysis prototypes (functional design stage) determine feasibility of a COTS-based solution and feedback to the requirements definition (definition and analysis stage). Design prototypes (physical design stage) provide hands-on experience with potential COTS products and feedback to the COTS product selection process (functional design stage). Detailed design prototypes (physical design stage) exercise functionality of selected COTS products, verify adherence and consistency with design expectations, reveal detailed behavior and performance characteristics, and give insight into the invocation parameters. The demonstration prototype (construction and test stages) is used to unit-test the COTS products using black-box testing to simulate application behavior or environment. Each stage is a potential source of feedback to previous stages.

3.3. Practical Considerations

The following practical considerations were encountered during 2 years of experience using the IIDA:

- The COTS product integrator does not develop the COTS product but still must know it internally. The integrator must understand the complete set of capabilities provided by the COTS product in order to select the appropriate subset of capabilities based on application developer needs for a given release of infrastructure. The integrator must understand the limitations and nuances of the COTS product in order to exercise it. For example, does it run on all of the required platforms? Does it operate the way it is intended? Does it have a heritage from a different paradigm (PC vs. UNIX workstation)?
- The system administrators and configuration management staff need to know how to configure the COTS products. Few complex COTS products work "out of the box." To support early prototypes and evaluations, not only do the designers and developers need to understand the products, but the development system administrators need to understand how to install and manage the product configuration. In addition, configuration management needs to understand how to configure the product versions.
- "COTS castles are often built on the sand of configuration files." Configuration files and data can be as complex as code. They must be understood. For example, a transaction processing monitor configuration file is inherently complex; training is required to know how to use it. Configuration files can be site-specific and require a strategy for managing files for different sites including site-specific parameters, implementation requests, and file distribution.
- When installing infrastructure components in new sites, the following documents that are not part of normal life cycles are critical for the configuration of COTS products:
  - — Release Notes (installation guidelines, operational parameters, tuning guidelines, etc.)
  - — Site Configuration Guidelines (guidelines to help site designers choose appropriate hardware and software suites, and rules for scaling and resource allocation).
- Version compatibility between COTS products, the operating system, and glue code is critical. This also applies to different sites including the external integration and test function. Software problems and nuances of use discovered during integration are not necessarily embedded in selected COTS products but often derive from specific characteristics of operating system versions or communications protocols. If application developers, infrastructure developers, and test sites are allowed to independently manage their computing platform configurations (including operating system and data base management system), trouble-shooting infrastructure anomalies is extremely difficult.

- Licensing adds a dimension of complexity and needs to be worked early. Issues include the number and types of licenses required for the environment. Short-term COTS evaluation licenses need to be managed, and transition needs to be planned from evaluation to product license. Procurement of production licenses within government agencies can require a long lead time and needs to start early with the bill of materials (BOM).

3.4. Technical Management Considerations

The following considerations can be easily overlooked during the planning cycle:

- The development facility including hardware, development tools, and configuration management must be ready to go before the first COTS product arrives for prototyping. Facility readiness fuels the accelerated development that using COTS products can provide but moves the requirement for a fully implemented development facility to early in the effort. Determining COTS suitability requires a realistic target configuration with a strong system administration team in place from the start.
- The BOM represents the contract for COTS products and versions. It is required early for field development sites and is essential for successful deployment.
- Technology infusion occurs by virtue of COTS product upgrades whether it is planned or not. Product upgrades can occur during any phase of the life cycle. Allowing for technology infusion can exploit new potential products on the market.
- The investment in training is a significant, but often overlooked, cost of using COTS products. Management needs to plan for the expertise of individuals to be shared across organizations. In particular, field sites need training; especially in system administration.

4. Conclusion

Integration with COTS software products requires adjustment and accommodations to the development approach versus traditional software development. Preparations must be made to start prototyping and integration activities immediately to exploit COTS product advantages and accelerate development. Additional resources must be allocated for late in the development cycle to provide maintenance and support to the user community, i.e., the application developers.

5. Acknowledgment

The authors would like to acknowledge the contribution of Mr. David P. Maloney, a software development manager at TRW, to this article. Many of the insights in the application of IIDA resulted from his work.

6. References

[1]. Berson, A., Section 1.2.2, "Openness and Proprietary Standards," *Client/Server Architecture*, McGraw-Hill, Inc., 1992.

[2]. Cerutti, D. and Pierson, D., Chapter 2, "The Rise of Open Systems," *Distributed Computing Environments*, McGraw-Hill, Inc., 1993.

[3]. National Institute of Standards, Application Portability Profile, The U.S. Government's Open System Environment Profile OSE/1 Version 2.0, June 1993, NIST Special Publication 500-210.

[4]. Martin, James, *Information Engineering*, Prentice Hall, 1989.

[5]. Software Engineering Institute, "A Commercial/ Business Perspective," *Proceedings of the SEI/MCC Symposium on the Use of COTS in Systems Integration*, Special Report CMU/SEI-95-SR-007, p. 24, June 1995.

[6]. Software Engineering Institute, "Systems Architecture and COTS Integration," *Proceedings of the SEI/MCC Symposium on the Use of COTS in Systems Integration*, Special Report CMU/SEI-95-SR-007, p. 26, June.

[7]. Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings ICSE9*, IEEE Computer Society Press, 1987.

[8]. Boehm B.W., "A Spiral Model of Software Development and Enhancement," *Computer*, pp. 61-72, May 1988.

[9]. Cooper, R., and Fox, G. "Technical Strategies to Guide the Design of Distributed Information Systems," *SIG Technology Review*, TRW Systems Integration Group, Vol. 4, No. 1, Winter 1996.

[10]. Langley, R.J., "COTS Integration Issues, Risks, and Approaches," *SIG Technology Review*, TRW Systems Integration Group, Vol. 2, No. 2, pp. 4–14, Winter 1994.

A Software Development Process for COTS-based Information System Infrastructure

Greg Fox, TRW
Steven Marcom, TRW
Karen Lantner, EDS

NASA/Goddard Space Flight Center
Twenty-Second Annual Software Engineering Workshop
Greenbelt, Maryland
December 3-4, 1997

A System Engineer Looks at the Software Lifecycle

- Control (Waterfall) versus Cycle-time (Spiral)
  - How do we have efficient processes and still maintain control?
  - How do we control prototyping?
- Applications versus Infrastructure
  - Inherently different, so should we use the same process model?
- Code development versus COTS integration
  - They aren't really the same, so should the same steps, milestones, documents apply?

Introduction

- Target system is very large, multi-component, multi-site and heterogeneous
- Infrastructure supports business application developers
- Infrastructure is primarily COTS products
- The methodology and experiences developing the initial versions of the Infrastructure are described

Services View of Infrastructure

- Human Computer Interface
- Systems Management
- Security
- Workflow Management
- Telecommunications
- Data Interchange
- Transaction Processing
- Data Management
- Operating Systems
- [ref: NIST APP]

COTS Challenge

- What set of life cycle processes best support the integration of COTS products?
- 1995 SEI/MCC Symposium on Use of COTS in Systems Integration, "... new life cycle models for COTS integration projects are needed."
- What are the distinguishing attributes of COTS products that influence the life cycle?

COTS Distinguishing Attributes

- Completeness
- Incompatibility
- Inflexibility
- Complexity
- Transience

Completeness

- COTS products enter the life cycle early as complete products
  - When a COTS product arrives, the first task is to test and integrate it into the system
  - Design, construction and integration/test stages of lifecycle must be recast to accommodate

Incompatibility

- Gaps and overlaps of functionality
  - Gaps can be overcome by custom coded "wrapper" software
  - Gaps can be left to the application designers to fill in the holes
  - Overlaps between products can result in a lack of synergism, compatibility and interoperability

Inflexibility

- Requirements mismatch, COTS exhibits unacceptable behavior
  - Option 1: replace the COTS software
  - Option 2: devise a workaround
  - Option 3: change the requirements

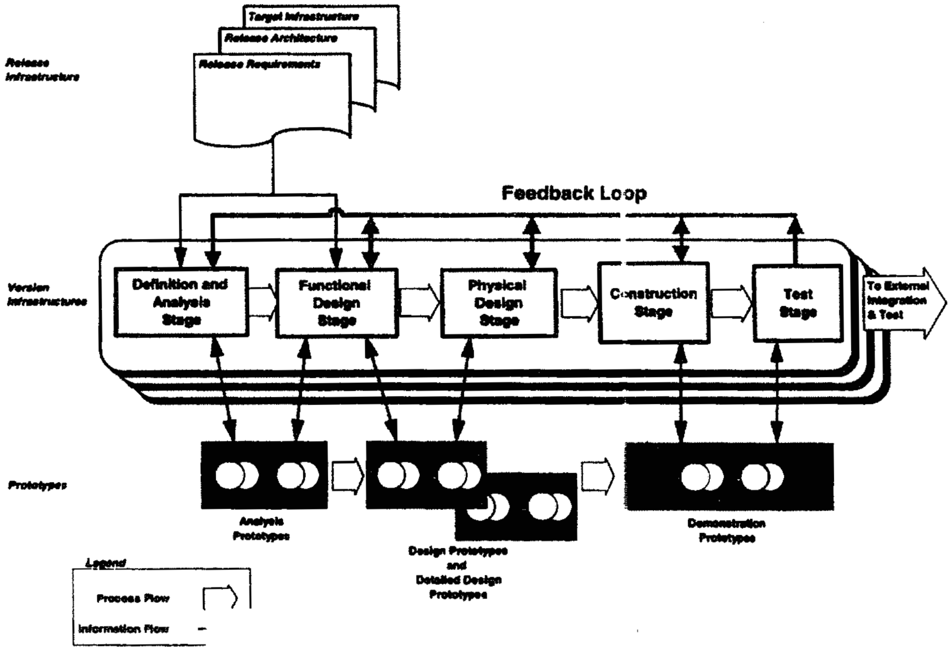
Complexity

- Long learning curves
  - Flexibility and tailorability of some infrastructure product families requires a significant education investment
  - Investment might have a net zero return if product proves unsuitable
  - Configuration of some COTS products can be as complex as code development with similar process needs

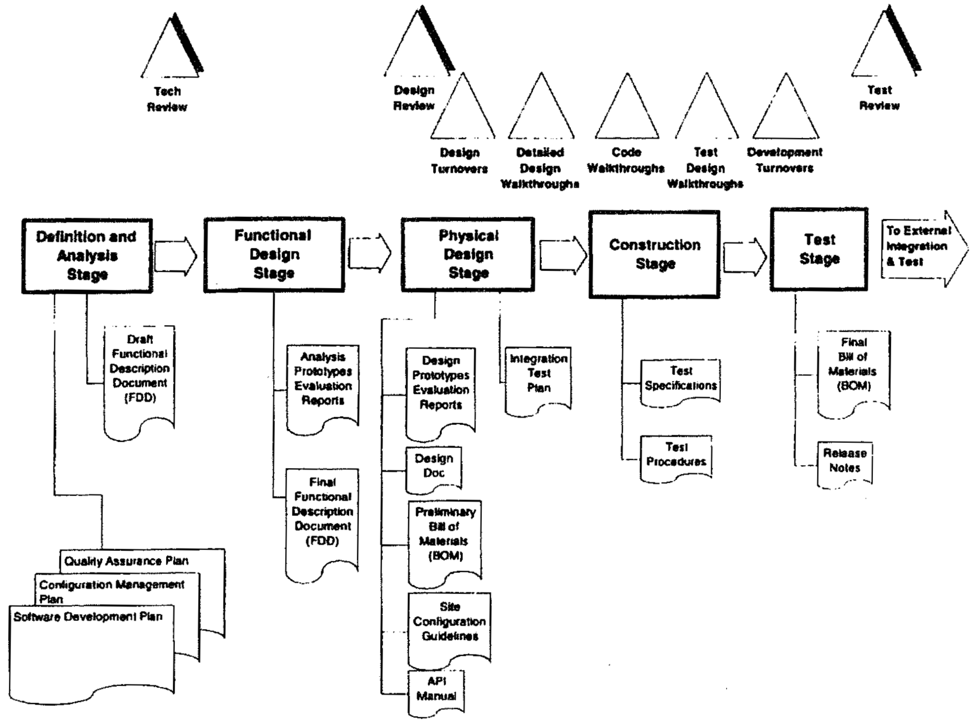
Transience

- Planned obsolescence
  - Transition to new COTS versions can be costly
    - Management
    - Direct and licensing costs
    - Technical factors
  - Interrelated products with different upgrade cycles can cause unexpected down time
- The following process model reflects the complexity of these characteristics

Infrastructure Incremental Development Approach (IIDA)



IIDA Milestones and Deliverables


Tech Review
Design Review
Test Review
Design Turnovers
Detailed Design Walkthroughs
Code Walkthroughs
Test Design Walkthroughs
Development Turnovers
Definition and Analysis Stage
Functional Design Stage
Physical Design Stage
Construction Stage
Test Stage
To External Integration & Test
Draft Functional Description Document (FDD)
Quality Assurance Plan
Configuration Management Plan
Software Development Plan
Analysis Prototypes Evaluation Reports
Final Functional Description Document (FDD)
Design Prototypes Evaluation Reports
Integration Test Plan
Design Doc
Preliminary Bill of Materials (BOM)
Site Configuration Guidelines
API Manual
Test Specifications
Test Procedures
Final Bill of Materials (BOM)
Release Notes

Analysis Prototypes

- Identify COTS products in COTS family
- Support FDD, Design Doc and API Manual
- Benefits
  - Determine gaps and overlaps among COTS
  - Uncover major COTS requirements disconnects
  - Understand functionality at a high level
  - Assess product stability, plans for new versions
  - Start Integration and Test early

Design Prototypes

- Evaluate and select best COTS candidate from Analysis prototype
- Support Design Doc, API Manual and BOM
- Benefits
  - Further analyze requirements satisfaction
  - Return to Analysis Stage, if needed
  - Identify workaround, if needed
  - Invest detailed knowledge of COTS operation

Detailed Design Prototypes

- Exercise COTS to verify consistency with design expectations, COTS documentation
- Support detailed design and Site Config
- Benefits
  - Identify low-level requirements mis-matches
  - Feedback to Design / Analysis Stages to modify or redesign with substitute COTS
  - Design low-level workarounds
  - Understand lowest level COTS complexity

Demonstration Prototypes

- Provide platform for CI unit test, integration
- Support test planning and Release Notes
- Benefits
  - COTS may not perform as expected or as documented
  - Feedback to functional design activity to modify or redesign with substitute COTS
  - Workarounds tested within COTS testbed

COTS Integration Pitfalls

- Integration and test begins with the arrival of the first COTS product
- Development/Test environment must be ready to support COTS integration and test
- Identification of COTS defects is costly
- Vendor maintenance turnaround can be long
- COTS integration and custom code development must proceed concurrently

IIDA Lessons Learned

- Prototype Engineers need early training
- System Administrators need early training
- CM must be organized early to support multiple versions of COTS products
- COTS configuration file development can be as complex as code development
- COTS licensing needs early planning

Summary

- Established process models for COTS integration are not yet available
- COTS integration is governed by particular characteristics that must be addressed by the development life cycle
- The IIDA is a successful approach to COTS integration which minimizes pitfalls

Author E-Mail Addresses

- greg.fox@ trw.com
- marcoms@gisdbbs.gisd.trw.com
- KLantner@aol.com

Page intentionally left blank

Page intentionally left blank

Experiences With CMM and ISO 9001 Benchmarks 360839

by
Joe Haskell, William Decker, and Frank McGarry
Computer Sciences Corporation (CSC)

Abstract

The use of industry benchmarks to measure process maturity and process compliance has increased significantly in recent years. Two widely applied benchmarks are the Software Engineering Institute's (SEI's) Capability Maturity Model (CMM) and a set of quality standards developed by the International Standards Organization (ISO). Although there is disagreement regarding the value and application of these benchmarks, many Government organizations use them to identify and select qualified contractors. Thus, it is becoming increasingly important for suppliers of products and services to become ISO registered and CMM compliant to satisfy criteria stipulated by potential customers.

CSC's SEAS Center attained ISO 9001 registration in May 1997. In November 1997, the SEAS Center was rated at CMM Level 3 based on a Software Capability Evaluation (SCE), with several Level 4-5 key process areas (KPAs) also satisfied. Information regarding the activities and effort to attain ISO registration and CMM Level 3 compliance was collected from SEAS Center participants. Also collected was participants' opinions regarding the impact that pursuit of each benchmark had on the organization. This information served as a basis for determining (1) the SEAS Center resources required to attain ISO registration and CMM Level 3 compliance and (2) the impact that pursuit of each of these industry benchmarks had on improving the SEAS Center's ability to deliver high-quality products and services to its customers.

This paper documents the experiences of the SEAS Center in pursuing CMM compliance and ISO 9001 registration. It is based entirely on information collected from SEAS Center personnel who planned and participated in the successful ISO 9001 registration and SCE efforts. Other organizations contemplating an SCE and/or ISO registration should find the experiences of the SEAS Center useful in planning their effort.

1. Introduction

The use of industry benchmarks to measure process maturity and process compliance has increased significantly in recent years. Two widely applied benchmarks are the SEI's CMM (Reference 1) and a set of quality standards developed by the ISO (Reference 2). Although there is disagreement regarding the value and application of these benchmarks, many Government organizations use them to identify and select qualified contractors. For example, NASA has adopted ISO 9000 as an internal requirement (Reference 3), and many of NASA's future contracts will likely require compliance with an applicable ISO standard. Similarly, other Government organizations require a specified CMM maturity level as a prerequisite for submitting a proposal. Compliance with industry benchmarks is becoming increasingly important to organizations seeking Government contracts.

CSC's SEAS Center provides support to NASA's Goddard Space Flight Center. As part of its endeavor to continually improve its process capabilities and comply with anticipated NASA requirements, the SEAS Center hired qualified external teams to evaluate its degree of compliance with CMM and ISO 9001. In May 1997, the Center achieved ISO registration, and a November 1997 SCE rated the SEAS Center as compliant with CMM Level 3.

The cost and implementation time for achieving compliance with each of these industry benchmarks are dependent on the organization's starting point. A mature organization will focus on adjusting existing processes rather than defining them. In contrast, a less-mature organization will initially focus on defining, documenting, and deploying key processes. The cost and implementation time needed to achieve compliance with a given CMM level or ISO standard are much less for an organization with a solid process foundation on which to build.

A profile of the SEAS Center is needed to place its experiences in context. The Center consists of approximately 800 individuals providing products and services to NASA. Activities include systems engineering, software development and maintenance, system integration, and system operations. The work includes mission design, control center development, advanced systems evaluation, data processing, and flight dynamics mission operations. Approximately 40 percent of the work consists of software development and maintenance. Fifteen to 20 projects were active during the 1997 assessments. The size of these projects varied from 5 to 100 personnel.

The SEAS Center began software process self-assessments and SCEs in 1991. Between 1991 and 1996, organizational policies, process documentation, and support tools were enhanced based on experience. As a result, by early 1996, mature processes had been documented and were routinely used throughout the Center. These processes included

- A set of Program Office Directives defining minimum requirements for project performance
- An earned-value reporting system that had been in routine use for approximately 15 years.
- An 12-chapter, system development methodology
- A comprehensive set of standards and procedures
- A set of handbooks and guidebooks
- Various project-level documents to address project-specific issues

2. Application of Industry Benchmarks

The history of the SEAS Center benchmarking activities is shown in Figure 1. Note that the efforts of the early 1990s were mostly based on the CMM, with the organization focused on improving software processes. As time passed, non-software processes became increasingly important, and the SEAS Center redirected its efforts to improving all processes. Compliance with an ISO standard was not established as an organizational goal until early 1996.

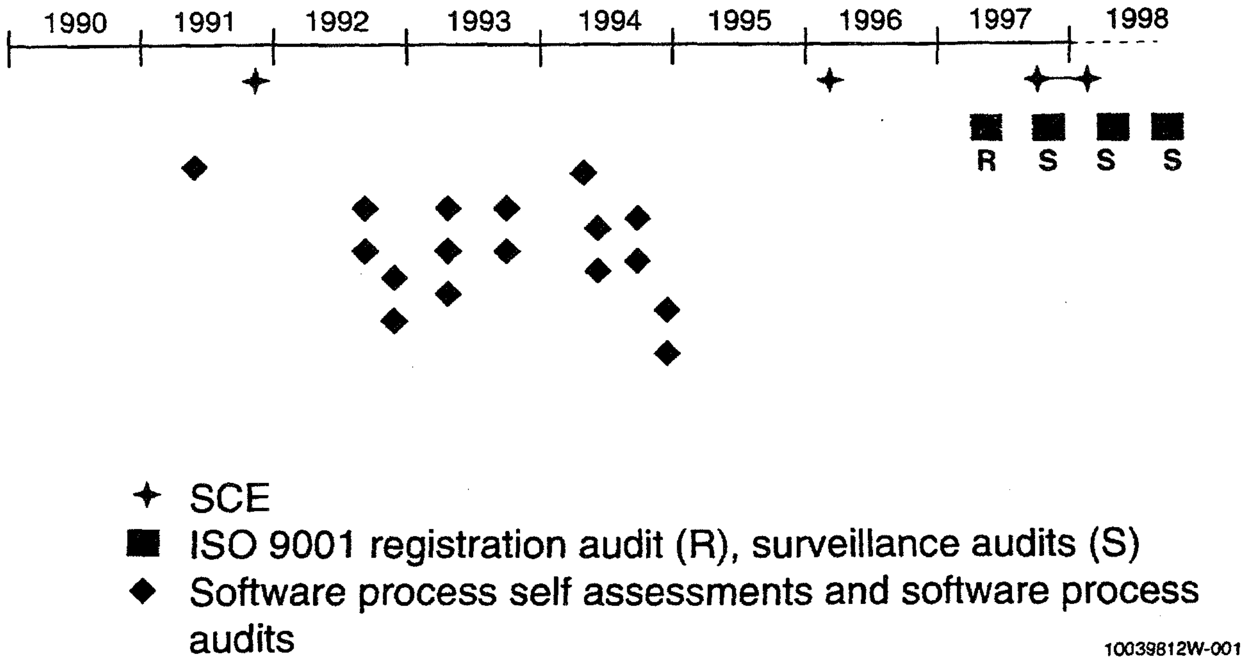


Figure 1. SEAS Center Benchmarking History

Information regarding each SCE and ISO 9001 assessment is summarized in Table 1. The 1991 SCE produced a CMM Level 1 rating for the SEAS Center, even though documented processes were routinely used. By February 1996, all process and process compliance weaknesses from the 1991 SCE had been addressed. As a result, all but one of the Level 2-3 KPAs were satisfied as determined by the February 1996 SCE. The SCE conducted in November 1997 found full compliance with all Level 2-3 KPAs, as well as compliance with several Level 4-5 KPAs.

The ISO registration assessment was conducted in May 1997. During the assessment, 14 minor nonconformities were identified, each of which was promptly eliminated. As a result, the ISO registrar recommended registration at the end of the assessment week. A subsequent surveillance assessment was successfully conducted in November 1997.

Since 1991, the SEAS Center has maintained detailed records regarding its benchmarking experiences. In addition to the summary information in Table 1, detailed records of all experiences were maintained. Based on these experiences, comprehensive lessons learned reports were prepared for use by other organizations pursuing ISO registration and CMM compliance.

3. Comparison of ISO 9001 and the CMM

Comparison of the CMM KPAs to the 20 elements of ISO 9001 has been performed by others (References 4 and 5) and is not the subject of this paper. Rather, this section provides a comparison of the ISO and CMM assessment processes, cost of each effort, and their relative value as perceived by the SEAS Center participants. It reflects the experiences of an organization that has applied both benchmarks.

Table 1. Summary of Benchmarking Activities

	SCE (10/91)	SCE (2/96)	ISO (5/97)	SCE (11/97 to 1/98)	ISO (11/97)
Preparation time	2 months	4 months	12 months	2 months	6 months
Organizational effort*	850 staff-hours	1800 staff-hours	3400 staff-hours	800 staff-hours	500 staff-hours
Use of external consultants and training	Minimal	None	• 200 hours • Consultant • Internal auditor training • Preregistration assessment	None	None
Preparation strategy	Perform software process assessments	• Perform gap analysis • Use lessons learned from 1991 SCE • Focus on deployment	• Develop implementation plan • Use external experts • Train staff • Focus on deployment	• Complete action items • Provide awareness seminars • Use internal assessments	• Continue process improvement initiatives • Focus on management review, internal audits, and corrective actions
Result	Few KPAs satisfied	13 of 18 KPAs satisfied	ISO registration achieved	CMM Level 3 achieved; Level 4-5 TBD 1/98	ISO registration maintained

* In addition to routine improvement activities

Organizations considering pursuing ISO registration and/or CMM compliance are usually interested in the answers to the following questions:

- How are the two benchmarks similar and different?
- For each, what is the cost of the effort and the implementation time?
- Is pursuit of ISO registration and/or CMM compliance worth the effort?

These questions are addressed in Sections 3.1 through 3.5.

3.1 Assessment Similarities

Based on the experience of the SEAS Center, the two types of assessment have the following similarities:

- **Selection of representative projects:** For each, a recommendation was made by the SEAS Center; the project selection was made by assessment team. For each type of assessment, the Center identified projects that, as a group, were representative of the work performed and covered the assessment criteria. The assessment team made the final selection, taking into account SEAS Center input.
- **Number of projects that represented the SEAS Center:** Four
- **Number of personnel interviewed during the assessments:** 50 to 60
- **Assessment duration:** 1 week
- **Degree of assessment focus on process definition, process compliance, and process improvement:** This was a key focus of both assessments.

3.2 Assessment Differences

Based on the experience of the SEAS Center, the two types of assessment have the following differences:

- **Scope of the assessment:** ISO 9001 covered all processes that affected product and service quality. The SCE was concerned only with software processes.
- **Documents reviewed by the evaluation team prior to the interviews:** The ISO team reviewed only the SEAS Center's *Quality Management System Manual* prior to the interviews. In contrast, the SCE team reviewed project profiles, completed CMM questionnaires, and 17 documents that had been mapped to the CMM.
- **Format of the interviews:** The ISO team interviewed members of each project, either individually or as a group. In contrast, the SCE team interviewed functional area representatives of all representative projects as a group (e.g., one software developer from each project)
- **Size of the evaluation team:** The ISO team had two members; the SCE team had six.

- **Evidence requested during the interviews:** The ISO team examined approximately 70 work products during the interviews. The SCE team examined approximately 100 work products, performing the examination after the interview.
- **Documentation of assessment results:** The ISO team documented assessment results in a short letter immediately following the assessment. The SCE team provided a comprehensive report documenting strengths, weaknesses, and improvement initiatives; the report was provided approximately 6 weeks following the SCE.

3.3 Activities, Required Effort, and Time to Prepare

Based on SEAS Center experience, the required effort and time to prepare for a successful evaluation depends on factors such as the following:

- Organizational experience with industry benchmarks
- Scope of evaluation (i.e., activities to be evaluated or KPAs to be examined)
- Degree of senior management commitment to a successful effort
- Maturity of organization's documented processes at the start of the effort
- Degree to which organization's defined processes are routinely used

The primary SEAS Center activities for its SCEs are defined in Section 3.3.1. The primary activities for ISO 9001 registration are defined in Section 3 3.2.

3.3.1 Software Capability Evaluation

As shown in Table 1, SEAS Center personnel spent 1800 staff-hours on the SCE conducted in February 1996. Most of this time was spent

- Identifying gaps in the organization's documented processes, relative to CMM Levels 2-3
- Updating documentation to improve processes and fill the gaps
- Deploying the revised processes (primarily training)
- Performing administrative work, such as completing the project profiles and CMM questionnaires required by the SCE team
- Participating in SCE interviews

Only 800 staff-hours were required to prepare for the SCE conducted in November 1997. By that time, SEAS Center personnel were confident that all CMM Level 3 processes were documented and routinely used. Note that because this SCE built on the experiences from the 1996 SCE, it required far less effort.

3.3.2 ISO Registration

As shown in Table 1, SEAS Center personnel spent 3400 staff-hours on the ISO registration effort conducted in May 1997. Most of this time was spent

- Gaining an understanding of ISO 9001 and preparing an implementation plan
- Performing a gap analysis based on ISO 9001
- Updating documentation to improve processes and fill the gaps (included writing a quality manual as required by ISO)
- Deploying the revised processes and support tools (e.g., Lotus Notes deployed to support internal communications and document control)
- Performing administrative tasks, such as selecting a ISO registrar (third-party external assessor)
- Participating in the ISO interviews

The ISO registration effort in May 1997 was successful. However, ISO requires periodic surveillance assessments to ensure process improvement is continuing. Only 500 staff-hours were required to prepare for the surveillance assessment conducted in November 1997. By that time, SEAS Center personnel were confident that all outstanding issues from the registration assessment had been adequately addressed. The surveillance assessment took place without noticeable impact on the organization.

3.4 Consultants and Trainers

Many consultants and trainers are available to assist an organization achieve ISO registration and/or CMM compliance. Regarding such services, the SEAS Center found the following:

- In 1997, typical cost of such services was $175 per hour, plus expenses.
- Companies and individuals providing such services seem well-qualified and competent. (It is a very competitive industry.)
- Such services can be worthwhile to establish credibility and minimize the possibility of false starts.

3.4.1 Software Capability Evaluation

The SEAS Center did not use consultants to prepare for its SCE. However, several SEAS Center personnel attended training in the CMM that was provided by its parent CSC organization.

3.4.2 ISO-Based Evaluation

The SEAS Center used consultants and trainers to prepare for ISO registration. Specifically, assistance was provided to the SEAS Center as follows:

- An individual from an external CSC division was assigned to the SEAS Center for several weeks at the start of the effort. This individual provided advice to the SEAS

Center ISO Implementation Team regarding implementation strategy. This individual also provided orientation and ISO awareness training for SEAS Center personnel. (total effort – 4 staff-weeks)

- An external trainer was hired to provide training of SEAS Center internal auditors. This individual also provided advice regarding other ISO-related activities. (total effort – 2 staff-weeks)
- The two-member ISO registration team conducted a preregistration assessment to familiarize SEAS Center personnel with the assessment process and identify any major weaknesses in the SEAS Center Quality Management System. (total effort – 1.5 staff-weeks)
- The two-member ISO registration team conducted the registration assessment. (total effort – 2 staff-weeks)

3.5 Value Based on Survey of Participants

The perceived value of the ISO registration and SCE experiences was measured by surveys of participants.

Immediately following ISO awareness training (1 year prior to the registration assessment), SEAS Center personnel were asked whether they felt the effort would improve the SEAS Center. Ten specific issues were addressed (e.g., improved processes, more customer confidence, less paperwork). Following the registration activity, the same personnel were asked whether they felt the effort did improve the SEAS Center. As shown in Figure 2, the ISO experience significantly exceeded expectations.

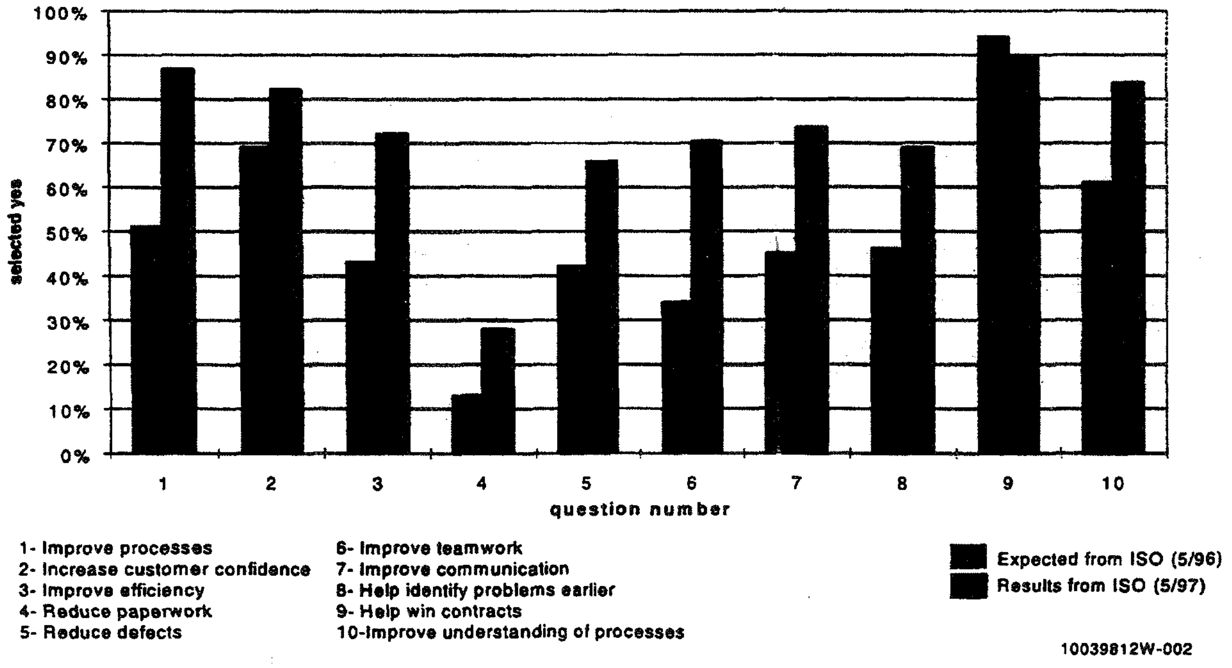


Figure 2. ISO Results Exceeded Expectations

The same issues were addressed by participants in the November 1997 SCE. The ISO experience was rated higher than the SCE on 8 of the 10 questions. A comparison of the answers to four of the questions is shown in Figure 3.

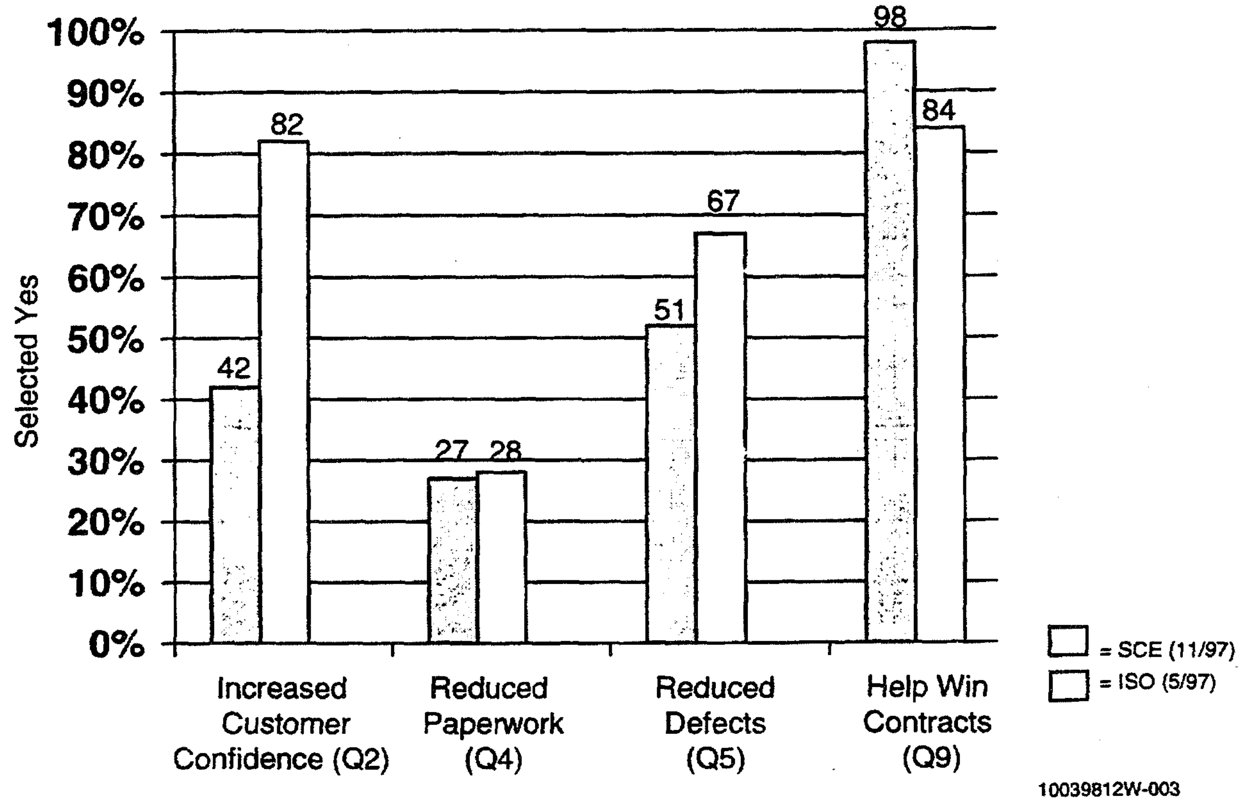


Figure 3. ISO 9001 Scored Higher than CMM on 8 Out of 10 Questions

Comments regarding these four answers follows:

- Increased Customer Confidence: ISO = 82%, SCE = 42%. This likely reflects the fact that the customer (NASA) has made ISO compliance a priority, while placing little value on CMM compliance.
- Reduced Paperwork: ISO = 28%, SCE = 27%. SEAS Center personnel found neither benchmark had much impact on the magnitude of process-related documentation and paperwork associated with process use.
- Reduced Defects: ISO = 67%, SCE = 51%. The relative low scores for each benchmark reflects minimal problems with product defects prior to benchmarking activities. (NASA's stated priority for the SEAS Center is cost and cycle time reduction; the quality of SEAS Center products has never been an issue.)
- Help Win Contracts: SCE = 98%, ISO = 84%. Government agencies other than NASA are increasingly using the CMM to identify and select contractors, hence the relatively

high score for SCE. Note that SEAS Center personnel strongly believe compliance with both industry benchmarks is important in obtaining new work.

SEAS Center personnel who participated in both assessments were asked: "In your opinion, which industry benchmark caused greater improvement." As shown in Figure 4, these individuals found ISO 9001 to have a greater positive impact. However, it is important to note that the ISO 9001 effort had an impact on the whole organization (rather than just software developers), and that the effort applied to ISO registration was almost twice as great as the effort applied to CMM compliance.

Finally, as shown in Figure 5, both benchmarks were viewed favorably by participants who were asked to rate the experience as (1) well worth the effort, (2) of marginal value, or (3) of little or no value. For both benchmarks, approximately 80 percent of the participants rated the activity as "well worth the effort." The percentage of favorable ratings by personnel not directly involved in the benchmarking activities was somewhat less than for participants.

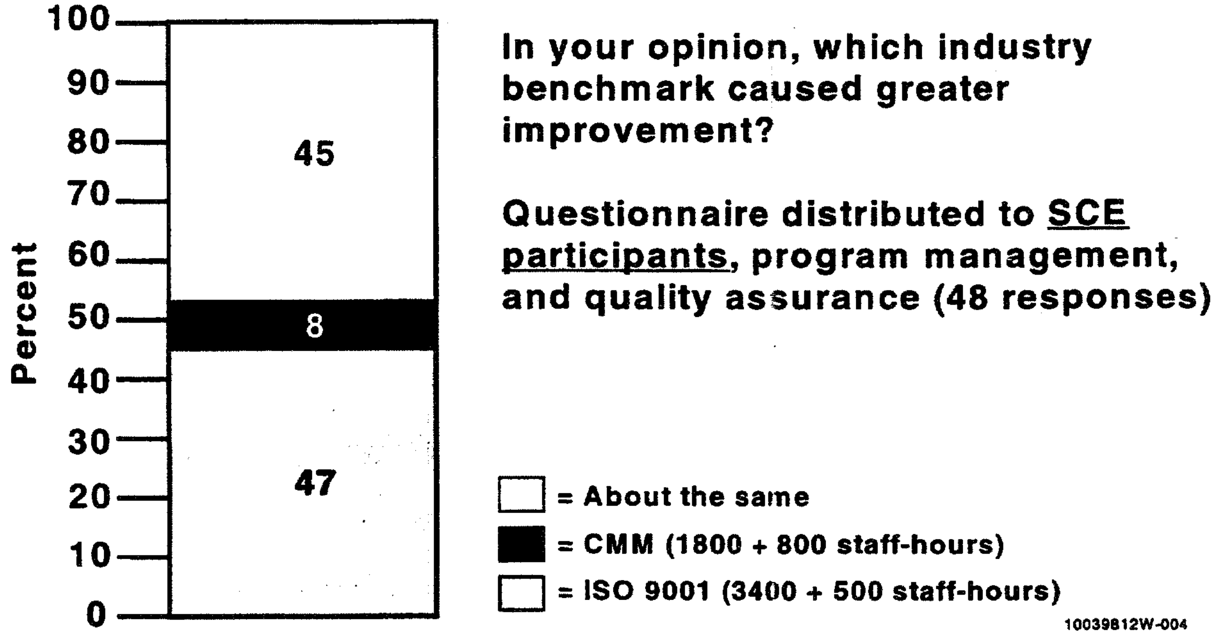


Figure 4. ISO 9001 had Greater Positive Impact than CMM, but also had Greater Effort Applied

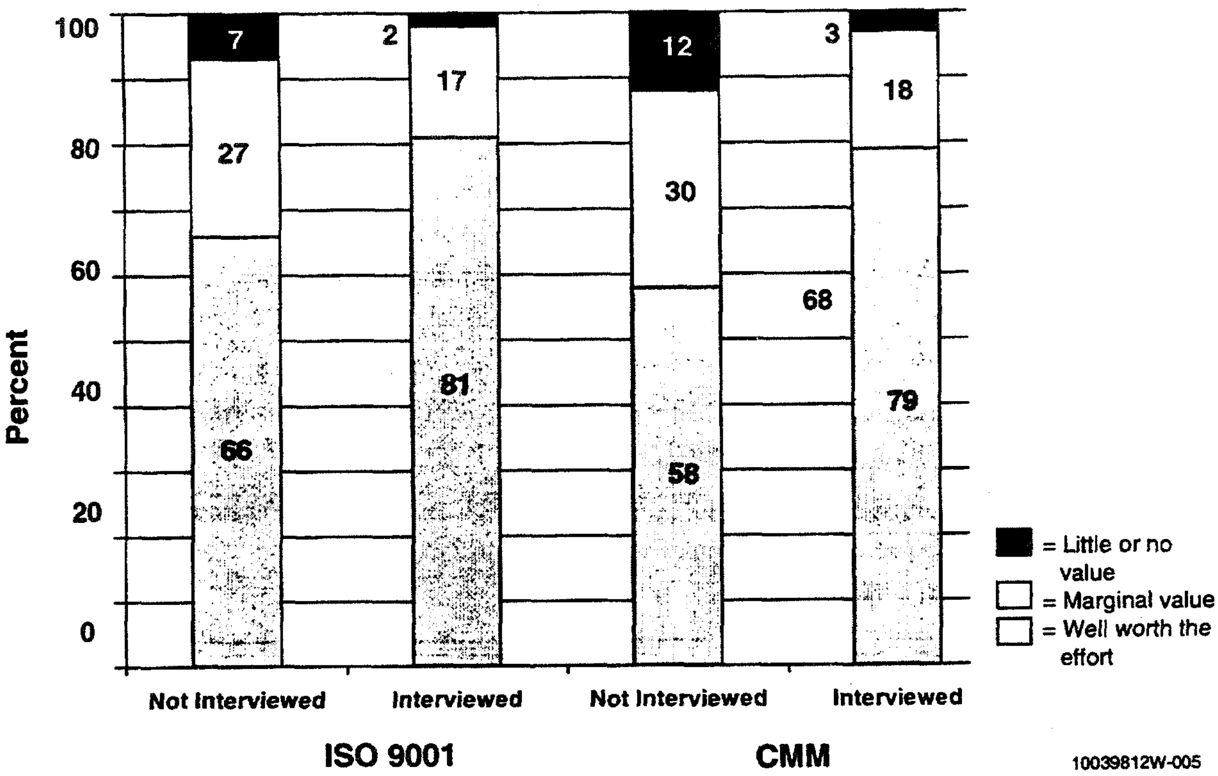


Figure 5. Both Benchmarks Were Viewed Favorably, Especially by Participants

4. *Impacts on SEAS Center*

Opportunities for organizational improvement as a result of a SCE or ISO Registration depend on the type of organization and its maturity. For the SEAS Center, many improvements resulted from the SCE and ISO registration activity. Some of these improvements are given below, with the primary impetus shown in parenthesis.

- Documented organization roles and responsibilities in the new *Quality Management System Manual* (ISO)
- Established a mechanism for document control (definition of current version, location, owner, and change process) (SCE)
- Established distribution of controlled documents through the use of electronic libraries (ISO)
- Refocused senior management reviews to address progress in achieving organizational goals related to products and processes (primarily SCE)
- Critically evaluated policies and processes to ensure effectiveness and alignment with organizational goals (prompted primarily by ISO)
- Revamped the training program to identify and address key organizational needs (primarily SCE)

5. Recommendations to Other Organizations

Based on the SEAS Center experience, the following recommendations should be considered by organizations setting out in pursuit of ISO registration and/or CMM compliance:

- Demonstrate commitment and participation of senior management; if the effort is not a high priority of senior management, it will fail.
- Establish an internal group responsible for interpreting the benchmark and providing guidance to the rest of the organization; do not require everyone to be well versed in CMM or the ISO standard.
- Implement processes and procedures that will improve the organization, rather than just comply with the industry benchmark.
- Do not install processes developed externally; build on what exists and use experiences of organizations doing similar types of work. (Installing new processes will not change the organization's culture.)
- Focus on compliance with the goals of the benchmark, rather than details.
- As a guideline, spend one third of the total effort developing and documenting new and revised processes and two thirds deploying the processes.
- Conduct internal audits to identify areas of noncompliance and track resulting action items to closure.

6. References

1. Charles V. Beber, et al., CMU/SEI-93-TR-24, ESC-TR-93-177, *Capability Maturity Model for Software*, Version 1.1, February 1993

2. *American National Standard. Quality Systems – Model for Quality Assurance in Design, Development, Production, Installation, and Servicing*, prepared by the American Society of Quality Control Standards Committee for American National Standards Committee Z-1 on Quality Assurance, August 1, 1994

3. NASA Management Instruction 1270.3 (12/6/95) and Goddard Policy Directive 2600 (6/18/97)

4. M. Paulk, "How ISO 9001 Compares With the CMM," *IEEE Software*, January 1995, pp. 74-82

5. F. Coalier, "How ISO 9001 Fits Into the Software World," *IEEE Software*, January 1995, pp. 98-100

Experiences with CMM and ISO 9001 Benchmarks

December 3, 1997

Joe Haskell
Frank McGarry
Bill Decker

Computer Sciences Corporation
SEAS Center

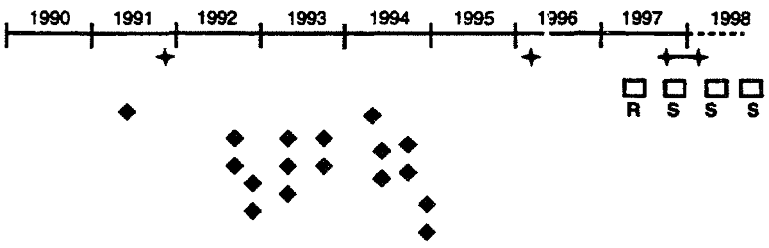
SEAS Center Profile

- SEAS Center has 800 personnel, primarily working on a NASA contract
  - 250 in software development and maintenance
  - 550 in systems engineering, analysis, and operations
- 20 software projects
- 10-year legacy of process improvement focus
  - First SCE in 1991
  - ISO 9001 registration in 1997

SEAS Center Process Improvement

- Following formal process improvement plan that specifies six measurable goals
  - One goal is ISO 9001 registration and compliance with CMM Level 3 as determined by an SCE
  - Other goals concern client satisfaction, technology infusion, product quality, productivity, and predictability
- In May 1997, registered to ISO 9001 standard
- In November 1997, assessed as CMM Level 3 by SCE (Levels 4 and 5 will be evaluated in January 1998)
- Records of investment, changes, approach, and impacts were recorded to benefit CSC organizations and share experiences with professional community

SEAS Center Benchmarking History



✦ SCE

□ ISO 9001 registration audit (R), surveillance audits (S)

◆ Software process self assessments and software process audits

Summary of Benchmarking Activities

	SCE (10/91)	SCE (2/96)	ISO (5/97)	SCE (11/97 to 1/98)	ISO (11/97)
Preparation time	2 months	4 months	12 months	2 months	6 months
Organizational effort*	850 staff-hours	1800 staff-hours	3400 staff-hours	800 staff-hours	500 staff-hours
Use of external consultants and training	Minimal	None	200 hours • Consultant • Internal auditor training • Preregistration assessment	None	None
Preparation strategy	Perform SPAs	• Perform gap analysis • Use lessons learned from 1991 SCE • Focus on deployment	• Develop implementation plan • Use external experts • Train staff • Focus on deployment	• Complete action items • Provide awareness seminars • Use internal assessments	• Continue process improvement initiatives • Focus on management review, internal audits, and corrective actions
Result	Few KPAs satisfied	13 of 18 KPAs satisfied	ISO registration achieved	CMM Level 3 achieved; Levels 4 and 5 TBD 1/98	ISO registration maintained

* In addition to routine improvement activities

Similarities Found in CMM and ISO Registration

- Same number of projects sampled (4)
- Same number of individuals interviewed (50 to 60)
- Same duration of evaluation (1 week)
- Same focus on process improvement
- Same focus on "do what you say"

Differences Found in CMM and ISO Registration

	SCE	ISO 9001 Assessment
Organization elements that participated	Program management, software projects, and support organizations	Program management, all projects, and support organizations
Materials requested in advance by evaluation team	Completed organization and project profiles, questionnaires, key software documents, and map of documents to CMM	Quality Management System Manual, definition of scope of registration
Interview format	Groups of functional area representatives from each participating project interviewed by entire SCE team	Project representatives interviewed individually or in a group
Size of evaluation team	Six people	Two people
Evidence requested	About 100 software-related work products; evidence typically gathered following the interview	About 70 work products; evidence typically examined during the interview
Result of assessment	CMM maturity level; strengths, weaknesses, and improvement initiatives defined in comprehensive report	Registered or not registered; weaknesses documented

Effort Distribution for CMM and ISO Registration was Similar

Activity	Primary CMM Activities	CMM (%)	ISO 9001 (%)	Primary ISO 9001 Activities (Registration)
Planning	• Management buy-in • Implementation plan	10	5	• Management buy-in • Implementation plan
Training	• CMM training • Project awareness	5	2	• Awareness training • Internal auditor training
Gap analysis	• KPA-by-KPA analysis	10	5	• Element-by-element analysis
Fill documentation gaps	• Update S&Ps and handbooks for organization and projects	15	1	• QMS manual • New or updated documentation
Deploy new or revised policies and procedures	• New process training • Evidence identification and gathering	45	3	• Electronic libraries • New process training • Evidence identification and gathering
Internal audits	• QA audits • Project internal audits	5	10	• QA audits • Preregistration assessment
Logistics	• Complete profiles and questionnaires • Gather organization and project documents • Participate in assessment	10	5	• Select registrar • Participate in assessment

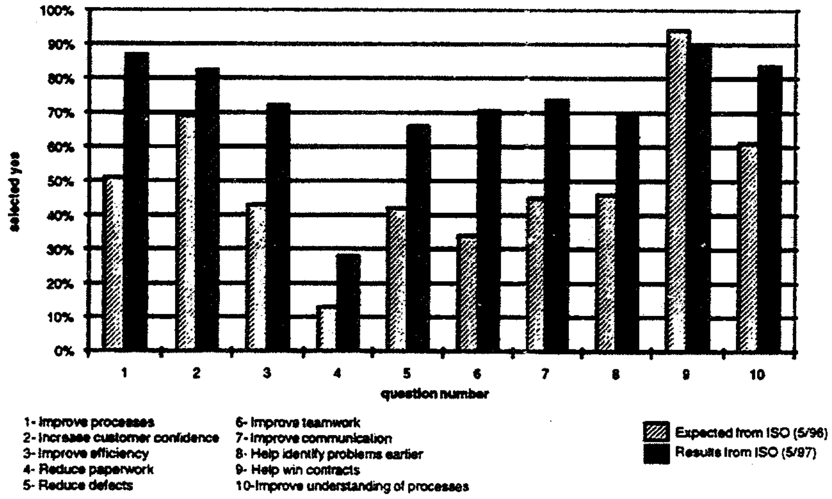
Benchmarks had Value to the SEAS Center

- Increased focus on achieving organizational goals (began operating as an enterprise)
- Improved communication, teamwork, and understanding and use of organizational processes
- Provided *hammer* for accelerating improvement programs
- Resulted in updating of policies and processes to address real needs of organization
- Accelerated adoption of technology across organization (e.g., electronic document libraries)
- Resulted in business advantage (increased credibility in proposals)
- Fostered pride in achieving one of organizational goals

Benchmarks Were Not a *Silver Bullet*

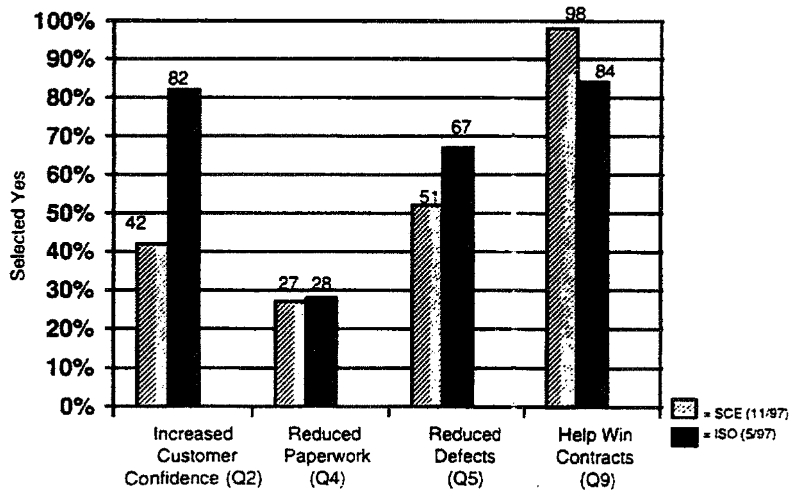
- Cost
- Temporarily diverts attention from project activities to compliance with benchmark
- No evidence of short term return on (process) investment for organization
- Tax on short-duration projects

ISO Results Exceeded Expectations*



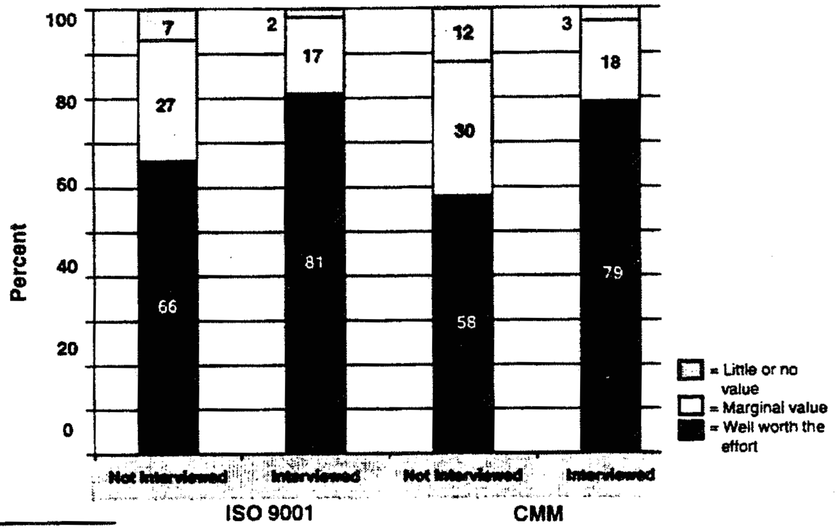
*Sample of 134 SEAS Center personnel

ISO 9001 Scored Higher than CMM* on Eight Out of Ten Questions



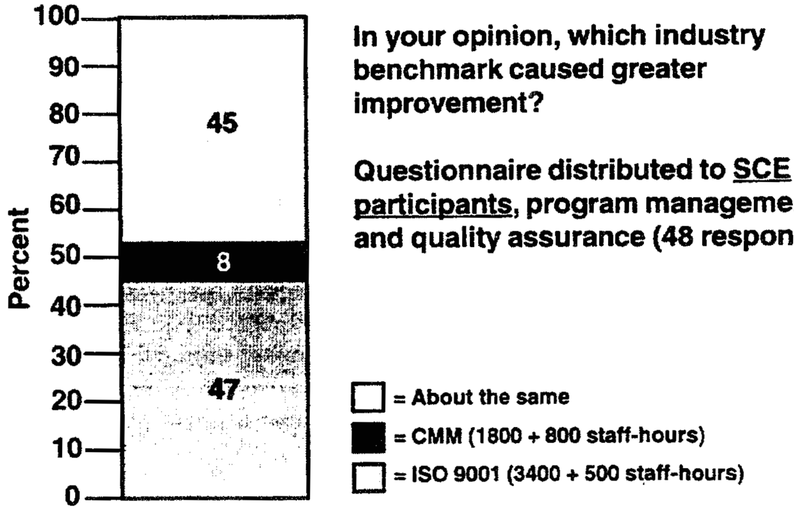
*Sample of 48 SEAS Center personnel who were SCE (and ISO) participants

Both Benchmarks Were Viewed Favorably, Especially by Participants*



*Survey based on random survey of 134 for ISO and 48 for SCE

ISO 9001 had Greater Impact than CMM, but also had Greater Effort Applied



In your opinion, which industry benchmark caused greater improvement?

Questionnaire distributed to <u>SCE participants</u>, program management, and quality assurance (48 responses)

Application of Benchmarks Produced Some Unexpected Results

- Did not produce cynicism across organization (both CMM and ISO 9001 readily accepted)
- Achieved benchmark compliance without adopting micro approach
- Amount of documentation unchanged (for ISO 9001 and CMM)
- Bureaucracy decreased and project managers given increased responsibility
- Little evidence of change in project's way of doing business
- CMM and ISO 9001 are consistent and complementary

Suggestions Based on SEAS Center Experiences

1. Demonstrate commitment of senior management
2. Establish a group responsible for interpreting the benchmark and providing guidance to rest of the organization; do not require everyone to be well versed in CMM or ISO standards
3. Implement process/procedures to improve your organization rather than just to fit industry benchmark
4. Do not use another organization's processes; build on what you have and use experiences of organizations doing similar types of work (documents do not change culture)
5. Focus on compliance with goals of the benchmark rather than details (e.g., CMM goals rather than activities)
6. Spend at least twice as much effort deploying than in documenting processes
7. Conduct internal audits and track action items to closure

/ 360846

p.

Using PSP and TSP Data to Manage Software Quality[1]

Watts S. Humphrey
The Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA
watts@sei.cmu.edu

In this paper, I briefly describe the Personal Software Process (PSP)SM and Team Software Process (TSP)SM. I then discuss the PSP/TSP quality strategy and show PSP data that demonstrate the logic for this strategy. Finally, I show preliminary data on how this strategy works in practice.

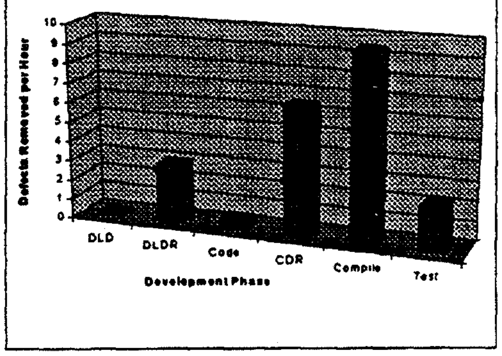
The Personal Software Process The PSP was developed by the SEI to help small software groups and organizations improve their performance. It provides a family of process scripts, forms, and standards that guide engineers through the steps of planning, tracking, and doing software work. The PSP is introduced with a textbook and course where engineers complete 10 programming exercises and 5 analysis reports [Humphrey]. The PSP is now being taught in a growing number of universities in the U.S., Europe, South America, and Australia, and it is being introduced by several software organizations.

The Team Software Process We developed the TSP because we found that many engineers had trouble applying the PSP to projects of more than one or two engineers. The TSP walks the team through launching and developing a product. While it uses the four generic phases of requirements, design, implementation, and test, its emphasis is on multiple product versions and interleaved activities. The TSP applies to teams of 2 to 20 hardware and software engineers who are PSP trained.

The TSP is introduced with a 3-day launch workshop where the engineers establish their goals, select their personal roles, and define their processes. They also make a quality plan, identify support needs, and produce a detailed development plan. The TSP shows them how to conduct a risk assessment, how to track and assess their work, and how to report their status to management. A 2-day relaunch workshop is used before each subsequent phase to integrate new team members, readjust role assignments, and reassess plans. The launch and relaunch workshops are not training courses; they are part of the project.

Some PSP quality data The PSP data provide some interesting information. We now have data on 2386 programs written in PSP courses. In all, the engineers took 15,534 hours to develop programs of 308,023 LOC. They found 22,644 defects. As shown in Figure 1, engineers find 9.48 defects/hour while compiling and in unit test they only find 2.21 defects/hour. During the PSP design (DLDR) and code (CDR) reviews, the engineers find 2.97 defects/hour and 6.52 defects/hour respectively.

Figure 1. Defects Removed per Hour



Engineers inject 1.76 defects per hour in detailed design and 4.20 defects per hour in coding. Thus, for every hour of design, an engineer should plan on at least 0.59 hours of design review to find all the design defects. Similarly, for every hour of coding, the engineer would need at least 0.64 hours of code review. Note that when engineers only use compile and test to remove defects, compile will generally find about 60% or more of the coding defects, thus reducing the minimum test time per hour of coding to about 0.74 hours.

The defect removal phases also inject defects. For example, in compile, engineers inject an average of 0.60 defects per hour while in test they inject an average of 0.38 defects per hour.

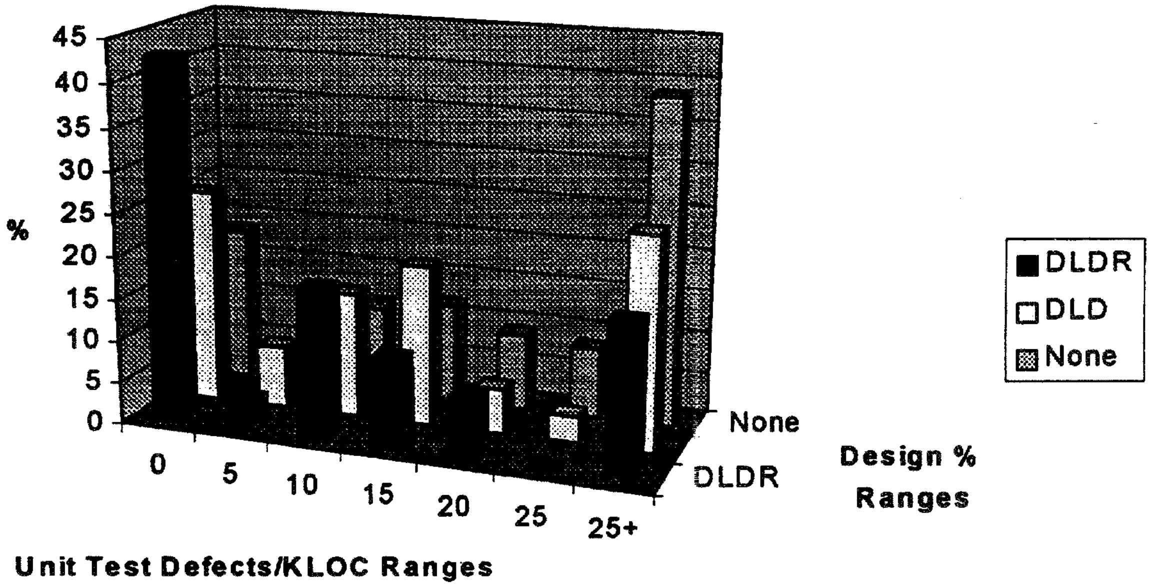
[1] This work is supported by the U.S. Department of Defense.

SM Personal Software Process and PSP are service marks of Carnegie Mellon University.

SM Team Software Process and TSP are service marks of Carnegie Mellon University.

Another interesting ratio is the number of defects injected per defect removed. In design review engineers inject a defect for every 27.9 defects removed while in code review they inject a defect for every 59.78 defects removed. In compile they inject a defect for every 15.84 defects removed and in test they inject a defect for every 5.82 defects removed. Note that before PSP training, these rates are significantly higher at one defect injected for every 9.54 defects removed in compile and one defect injected for every 4.43 defects removed in test.

Figure 2. Unit Test Defects vs. Design Practices



The PSP course data also show the value of using disciplined personal practices. When engineers spend an adequate amount of design time, they improve product quality. As shown by the middle bars in Figure 2, there are fewer test defects when engineers spend more time in design. The cutoff appears to be at about 50% of coding time. Similarly, when design time is over 50% of coding time and design review time is greater than 50% of design time, there are even fewer unit test defects. These values are shown in the front bars of the figure. As shown by the front bars in Figure 3, when code review times are greater than 75% of coding time, compile defects are also lower.

The PSP/TSP quality strategy The PSP/TSP quality strategy is illustrated by the case of a large IBM program shown in Figure 4 [Kaplan]. The correlation between development and usage defects is 0.964 for release one. For release two, the correlation was also high at 0.878. The number of defects found in development test thus seems to be a good indicator of the number of defects remaining after test. Therefore, to reduce usage defects engineers should remove defects before development test. This can only be done by having the engineers use a disciplined personal process. The PSP shows engineers how to remove defects at the earliest possible time, preferably before the first compile.

Industrial PSP data We are now getting early data on industrial use of the PSP and TSP. One set of data comes from Advanced Information Services (AIS) corporation in Peoria, IL. As shown in Table 1, their group in India shipped several components before their 12 engineers were PSP trained. After the 2-week PSP training course, the next release was nearly on schedule and had one acceptance test defect.

Figure 3. Compile Defects/KLOC vs. % Code Review/Code Time

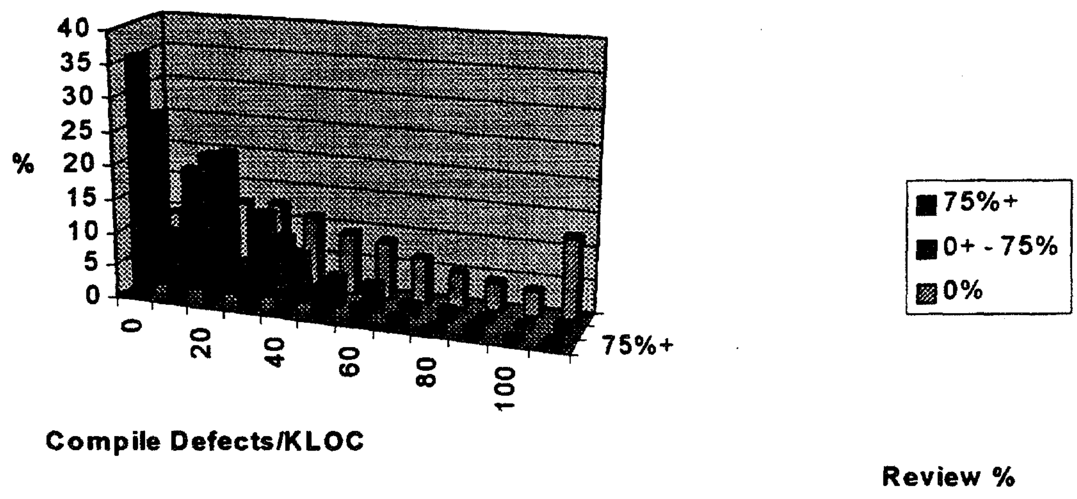


Figure 4. Development vs. Usage Defects - IBM Release 1 (r = 0.9844)

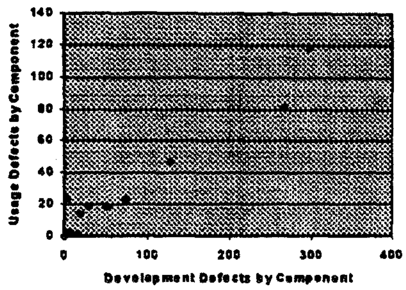


Table 1. AIS Data - PSP Impact on Acceptance Test Quality

Without PSP	KLOC	Months Late	Acceptance Defects
1	24.6	9	NA
2	20.8	4	168
3	19.9	3	21
4	13.4	8+	53
5	4.5	8+	25
With PSP	22.9	1	1

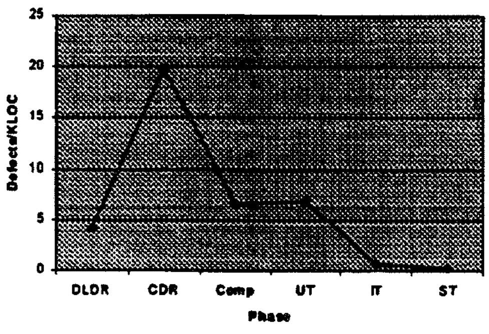
Table 2 shows the impact of improved PSP quality on system test time. Programs A1 and A2 were system tested together for 1.5 months.

Table 2. AIS System Test Time Reduction with PSP

Not using PSP	Size	Test Time
Project A1	15,800 LOC	1.5 months
Project C	19 requirements	3 test cycles
Project D	30 requirements	2 months
Project H	30 requirements	2 months
Using the PSP		
Project A2	11,700 LOC	1.5 months
Project B	24 requirements	5 days
Project E	2,300 LOC	2 days
Project F	1,400 LOC	4 days
Project G	6,200 LOC	4 days
Project I	13,300 LOC	2 days

TSP quality data While several industrial groups are using the TSP, the first completed project is from a team at Embry Riddle Aeronautical University (ERAU). Overall, product quality was good with 99.4% of the defects removed before system test. The team's defect removal profile is shown in Figure 5. Here, while the curve looks superficially good, there were some problems. The clue is that the number of design review defects (DLDR) is lower than the number found in unit test (UT). This indicates design review problems with at least some components. The TSP provides a quick way to identify defect-prone modules, as shown in Figure 6. This is the Defect Risk Profile for a component that had no defects found in integration (IT) or system (ST) test.

Figure 5. Defects/KLOC by Phase



The defect risk profile sets limits based on PSP data. Design time should be greater than 50% of coding time, design review time should be greater than 50% of design time, and code review time should be greater than 50% of coding time. While the data indicate that code review time should exceed 75% of coding time, we have been using 50% as the criteria, but will probably increase it. Also, compile defects should be less than 10/KLOC and unit test defects should be under 5/KLOC. When a factor meets or exceeds these criteria, that profile dimension is at the edge of the bullseye. When the criteria are not met, say 25% design review time instead of 50%, that dimension would be half way to the center of the bullseye. The profile in Figure 7 is for Component 9, which had one defect in integration test.

Figure 6. Component 7 Risk Factors

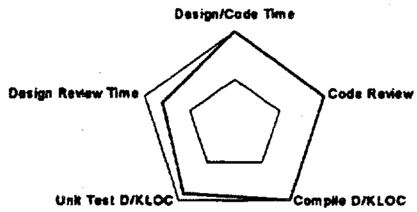
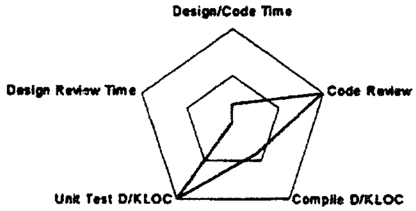


Figure 7. Component 9 Risk Factors



Defect risk profiles can be multiplied together to give a system profile. Then, a single high-risk component would result in a poor profile for the entire system. With the PSP and TSP data needed to make these profiles, it is easy to find the defect-prone modules, even in large systems with hundreds of modules and dozens of components. Note, however, that organizations should adjust these profile criteria based on their own data.

Conclusions The PSP and TSP provide extensive data that can be used to manage product quality. The TSP also shows engineers how to gather and use these data. While there are only limited industrial TSP data to date, early indications are that, with the PSP and TSP, engineers can produce very high quality programs at reduced costs and on competitive schedules.

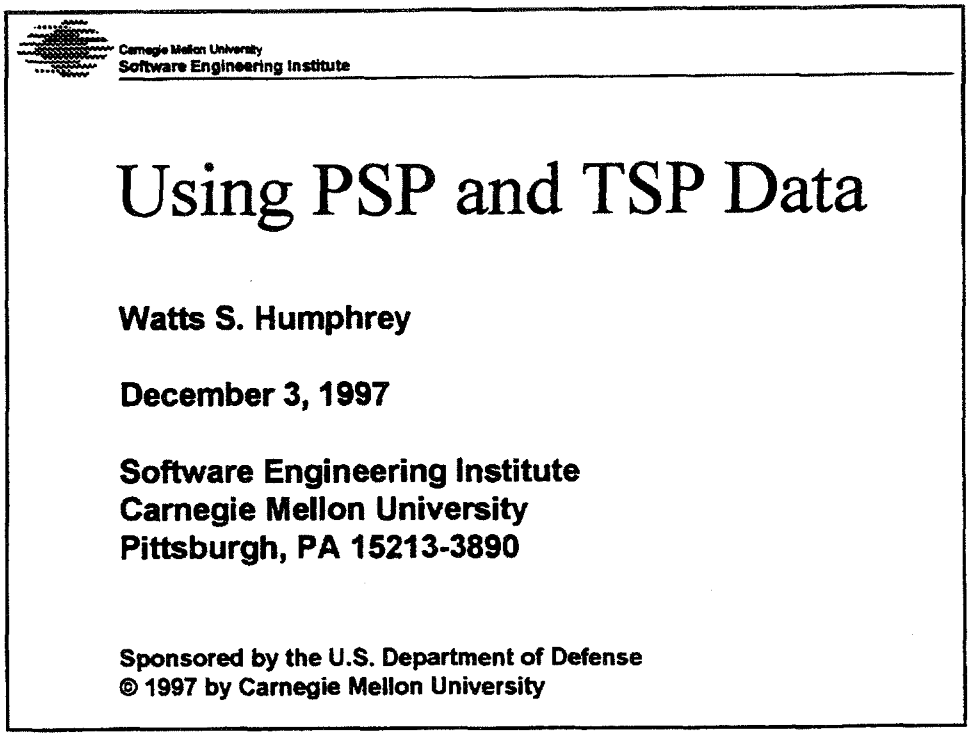
References

[Ferguson] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya, "Introducing the Personal Software Process: Three Industry Case Studies," *IEEE Computer*, vol. 30, no. 5, pp 24-31, May 1997.

[Humphrey] W. S. Humphrey, *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley, 1995.

[Kaplan] Craig Kaplan, Ralph Clark, and Victor Tang, *Secrets of Software Quality, 40 Innovations from IBM.* New York, N.Y.: McGraw-Hill, Inc., 1994.

Watts S. Humphrey
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Phone: 412-268-7701
Fax: 412-268-5758
e-mail: watts@sei.cmu.edu



Carnegie Mellon University
Software Engineering Institute

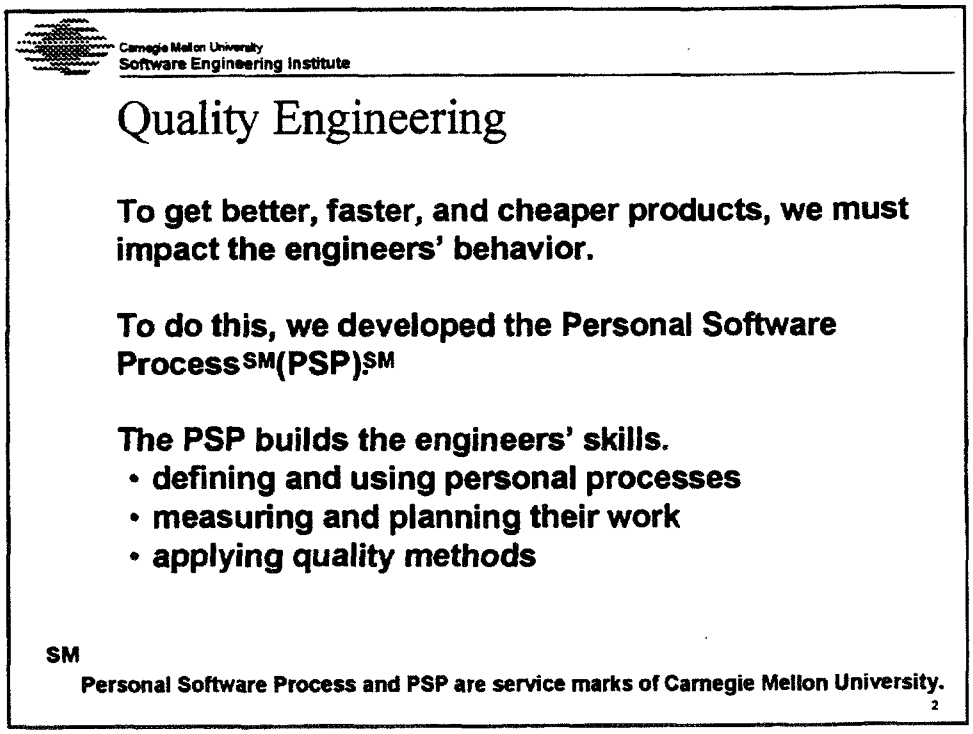
Using PSP and TSP Data

Watts S. Humphrey

December 3, 1997

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Sponsored by the U.S. Department of Defense



Carnegie Mellon University
Software Engineering Institute

Quality Engineering

To get better, faster, and cheaper products, we must impact the engineers' behavior.

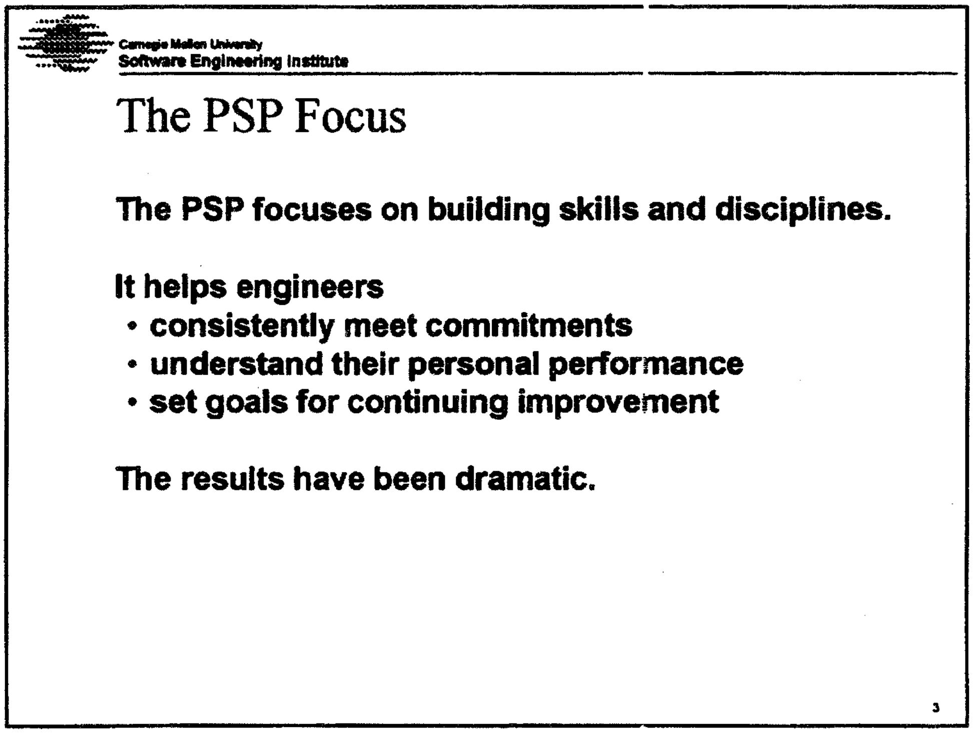
To do this, we developed the Personal Software Process^{SM}(PSP).^{SM}

The PSP builds the engineers' skills.

- defining and using personal processes
- measuring and planning their work
- applying quality methods

SM
Personal Software Process and PSP are service marks of Carnegie Mellon University.

2



Carnegie Mellon University
Software Engineering Institute

The PSP Focus

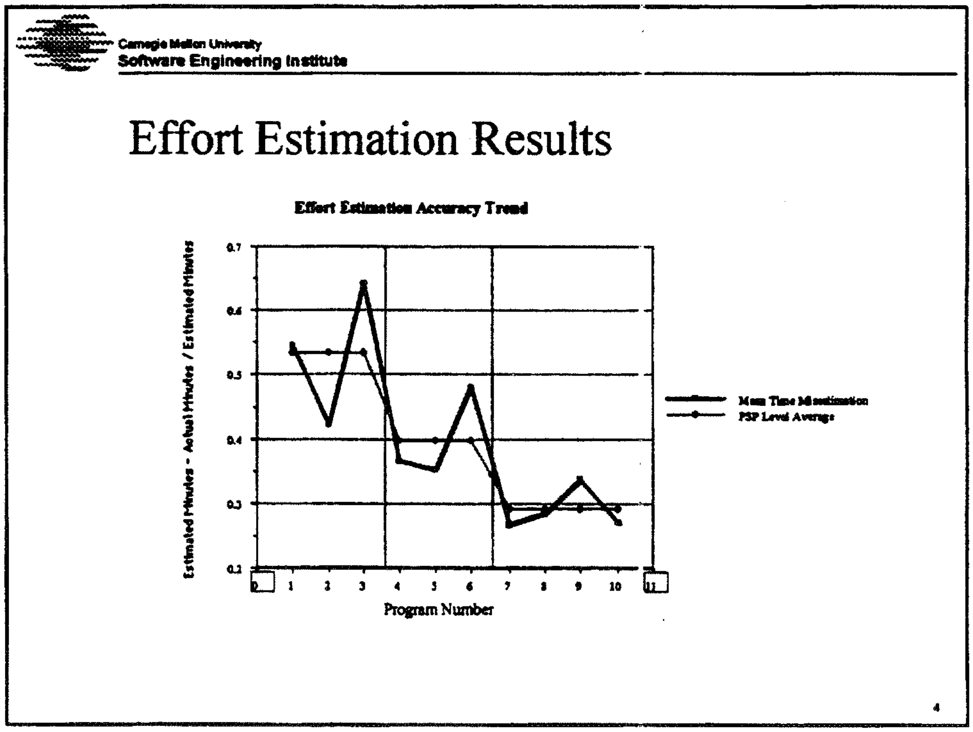
The PSP focuses on building skills and disciplines.

It helps engineers

- **consistently meet commitments**
- **understand their personal performance**
- **set goals for continuing improvement**

The results have been dramatic.

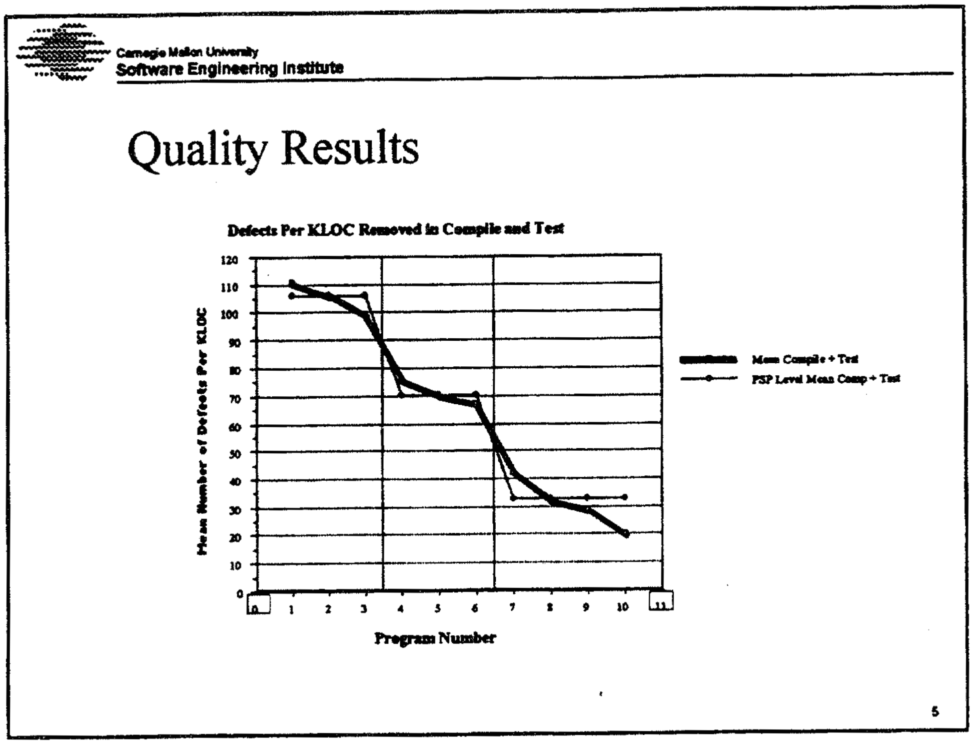
3



Carnegie Mellon University
Software Engineering Institute

Effort Estimation Results

4

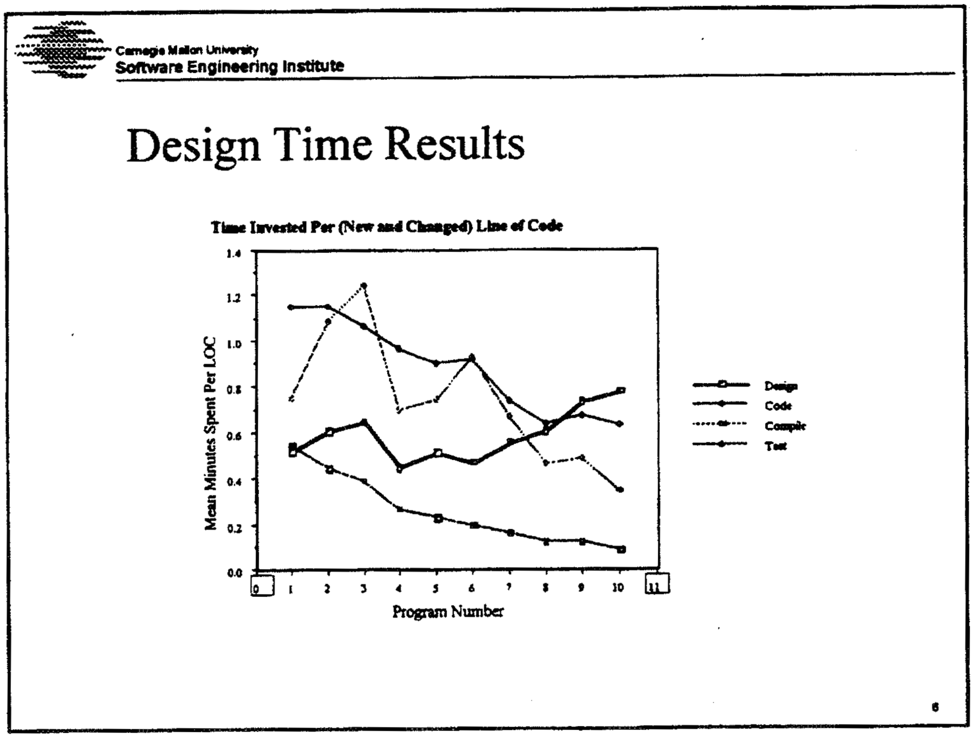


Carnegie Mellon University
Software Engineering Institute

Quality Results

Defects Per KLOC Removed in Compile and Test

5

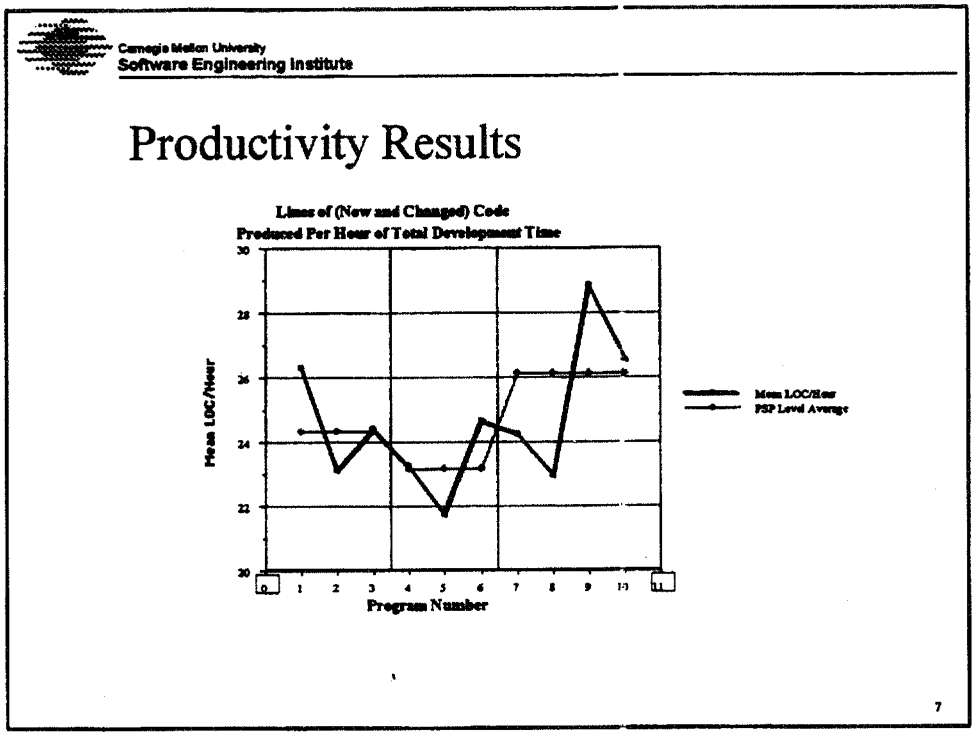


Carnegie Mellon University
Software Engineering Institute

Design Time Results

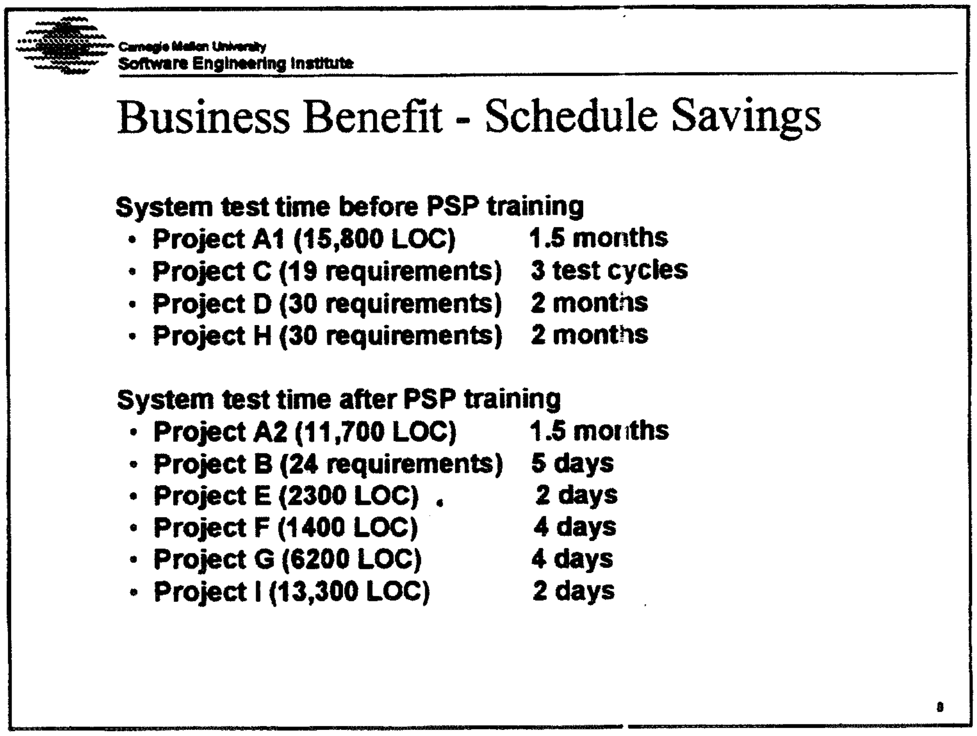
Time Invested Per (New and Changed) Line of Code

6


Carnegie Mellon University
Software Engineering Institute

Productivity Results

7


Carnegie Mellon University
Software Engineering Institute

Business Benefit - Schedule Savings

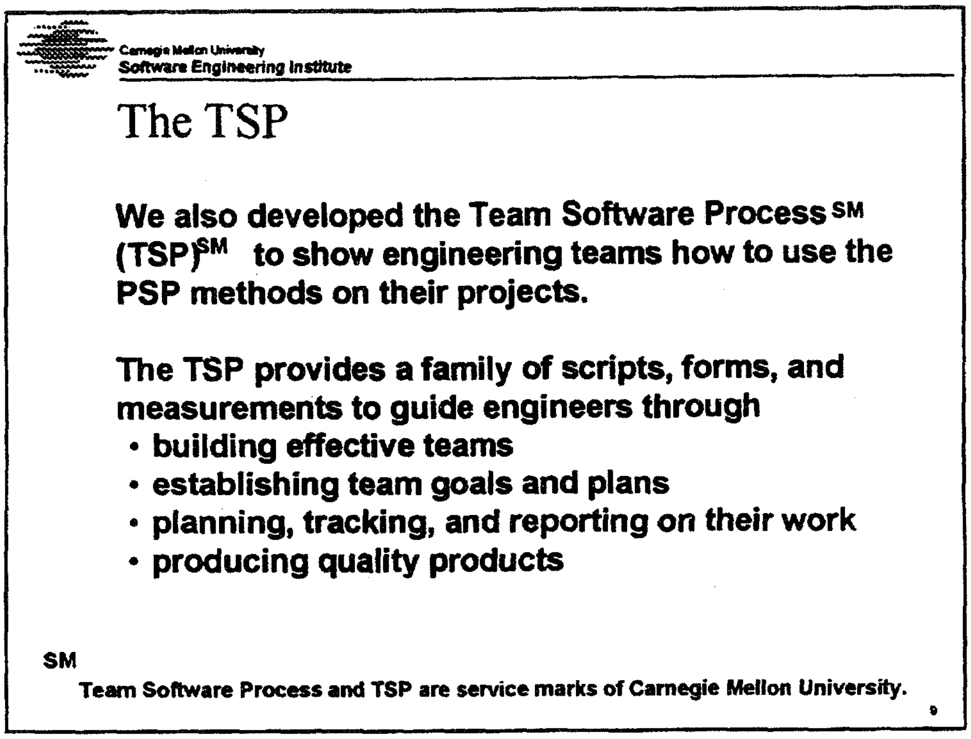
System test time before PSP training

• Project A1 (15,800 LOC)	1.5 months
• Project C (19 requirements)	3 test cycles
• Project D (30 requirements)	2 months
• Project H (30 requirements)	2 months

System test time after PSP training

• Project A2 (11,700 LOC)	1.5 months
• Project B (24 requirements)	5 days
• Project E (2300 LOC)	2 days
• Project F (1400 LOC)	4 days
• Project G (6200 LOC)	4 days
• Project I (13,300 LOC)	2 days

8



Carnegie Mellon University
Software Engineering Institute

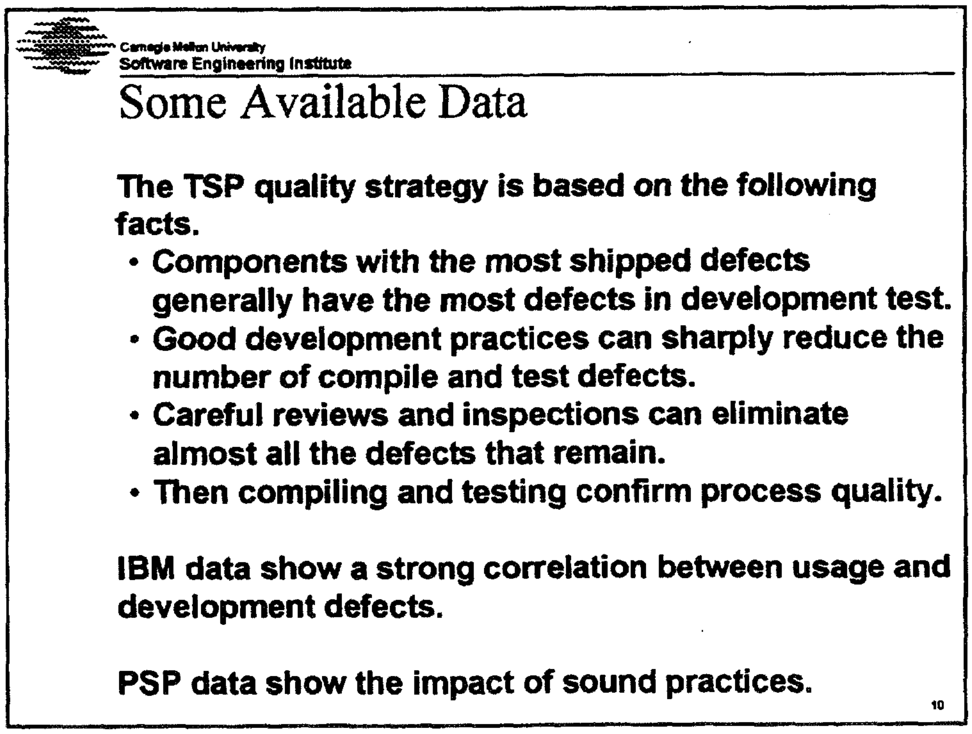
The TSP

We also developed the Team Software Process SM (TSP)SM to show engineering teams how to use the PSP methods on their projects.

The TSP provides a family of scripts, forms, and measurements to guide engineers through
- building effective teams
- establishing team goals and plans
- planning, tracking, and reporting on their work
- producing quality products

SM
Team Software Process and TSP are service marks of Carnegie Mellon University.

9



Carnegie Mellon University
Software Engineering Institute

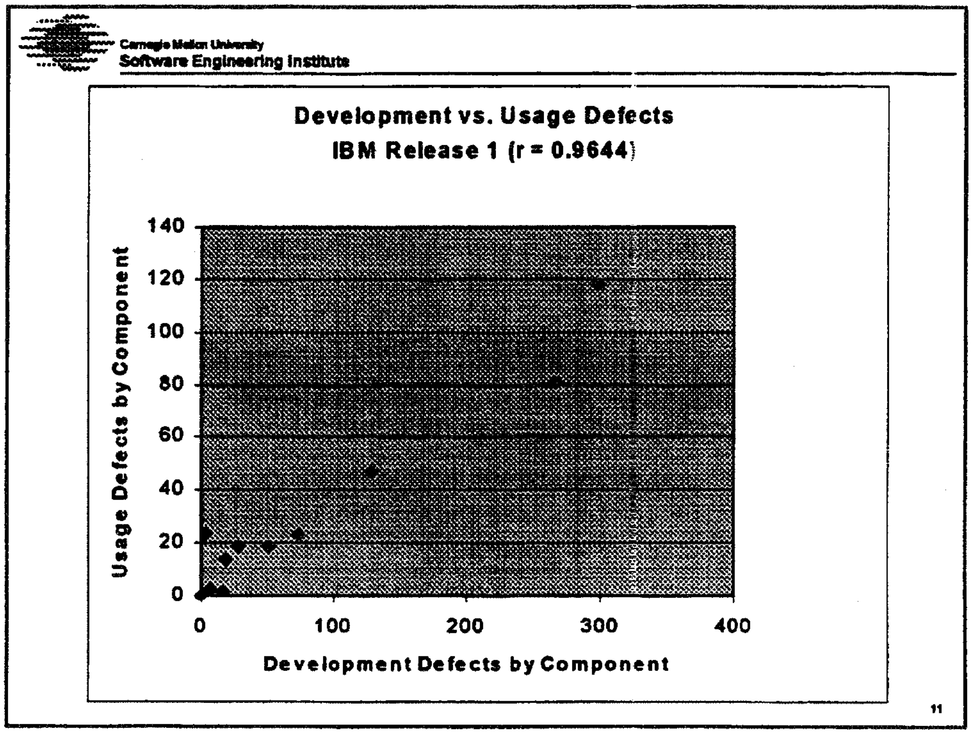
Some Available Data

The TSP quality strategy is based on the following facts.
- Components with the most shipped defects generally have the most defects in development test.
- Good development practices can sharply reduce the number of compile and test defects.
- Careful reviews and inspections can eliminate almost all the defects that remain.
- Then compiling and testing confirm process quality.

IBM data show a strong correlation between usage and development defects.

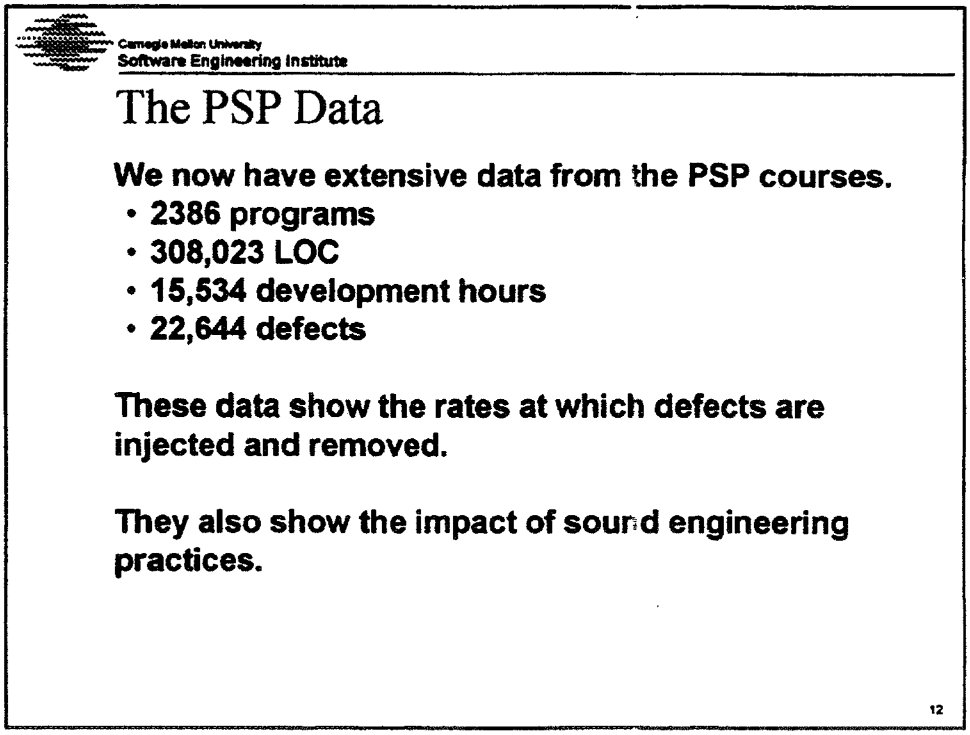
PSP data show the impact of sound practices.

10



Carnegie Mellon University
Software Engineering Institute

11



Carnegie Mellon University
Software Engineering Institute

The PSP Data

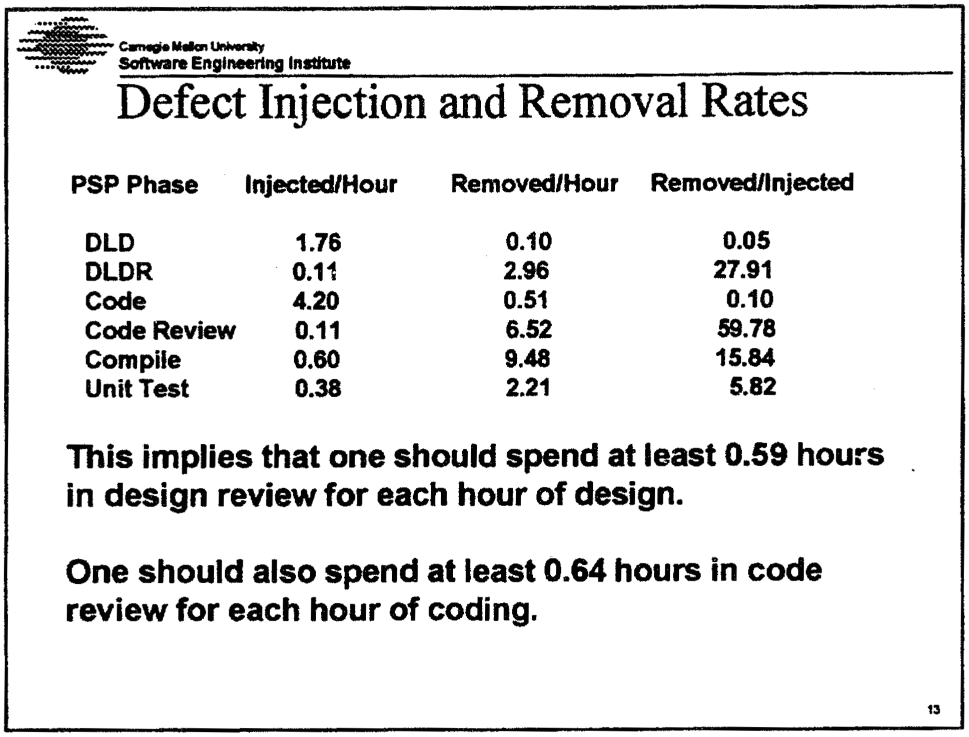
We now have extensive data from the PSP courses.

- 2386 programs
- 308,023 LOC
- 15,534 development hours
- 22,644 defects

These data show the rates at which defects are injected and removed.

They also show the impact of sound engineering practices.

12



Carnegie Mellon University
Software Engineering Institute

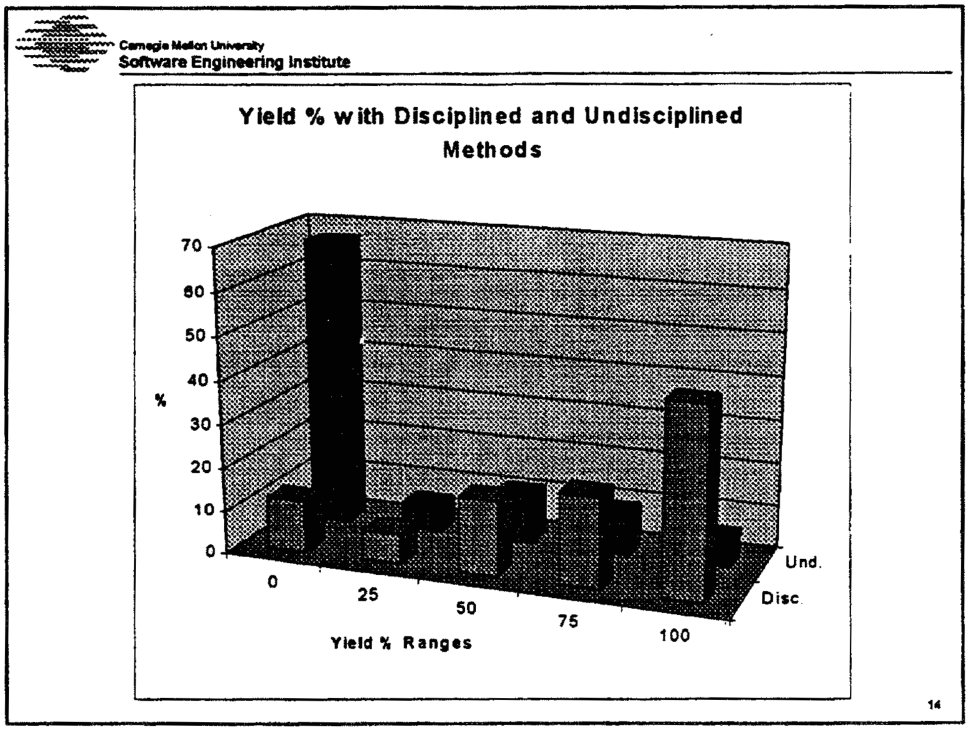
Defect Injection and Removal Rates

PSP Phase	Injected/Hour	Removed/Hour	Removed/Injected
DLD	1.76	0.10	0.05
DLDR	0.11	2.96	27.91
Code	4.20	0.51	0.10
Code Review	0.11	6.52	59.78
Compile	0.60	9.48	15.84
Unit Test	0.38	2.21	5.82

This implies that one should spend at least 0.59 hours in design review for each hour of design.

One should also spend at least 0.64 hours in code review for each hour of coding.

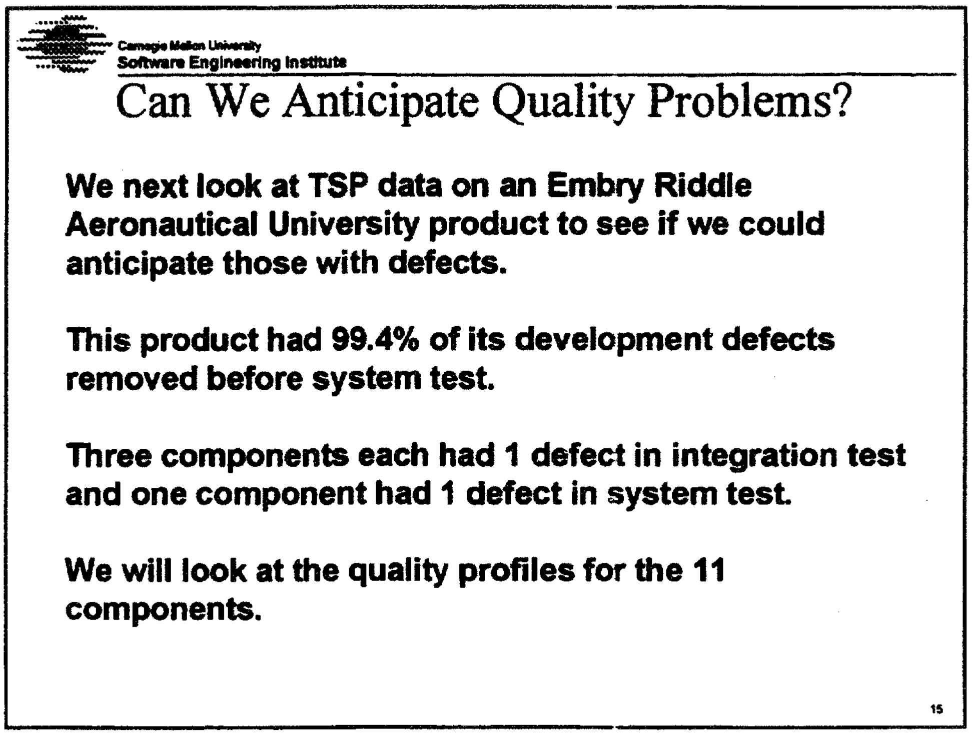
13



Carnegie Mellon University
Software Engineering Institute

Yield % with Disciplined and Undisciplined Methods

14



Carnegie Mellon University
Software Engineering Institute

Can We Anticipate Quality Problems?

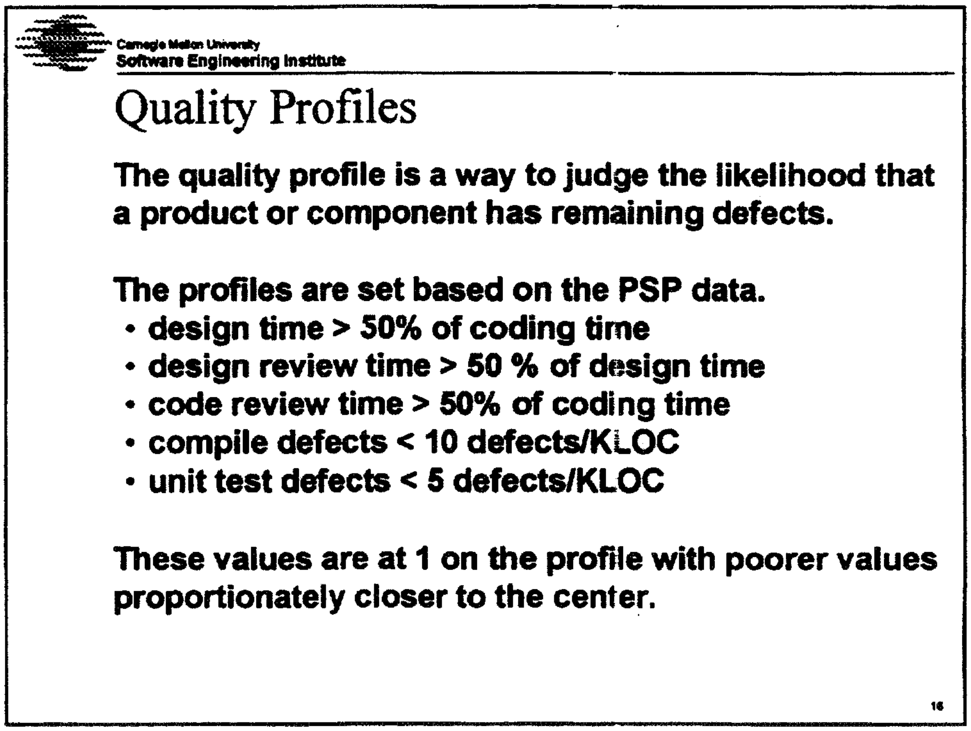
We next look at TSP data on an Embry Riddle Aeronautical University product to see if we could anticipate those with defects.

This product had 99.4% of its development defects removed before system test.

Three components each had 1 defect in integration test and one component had 1 defect in system test.

We will look at the quality profiles for the 11 components.

15



Carnegie Mellon University
Software Engineering Institute

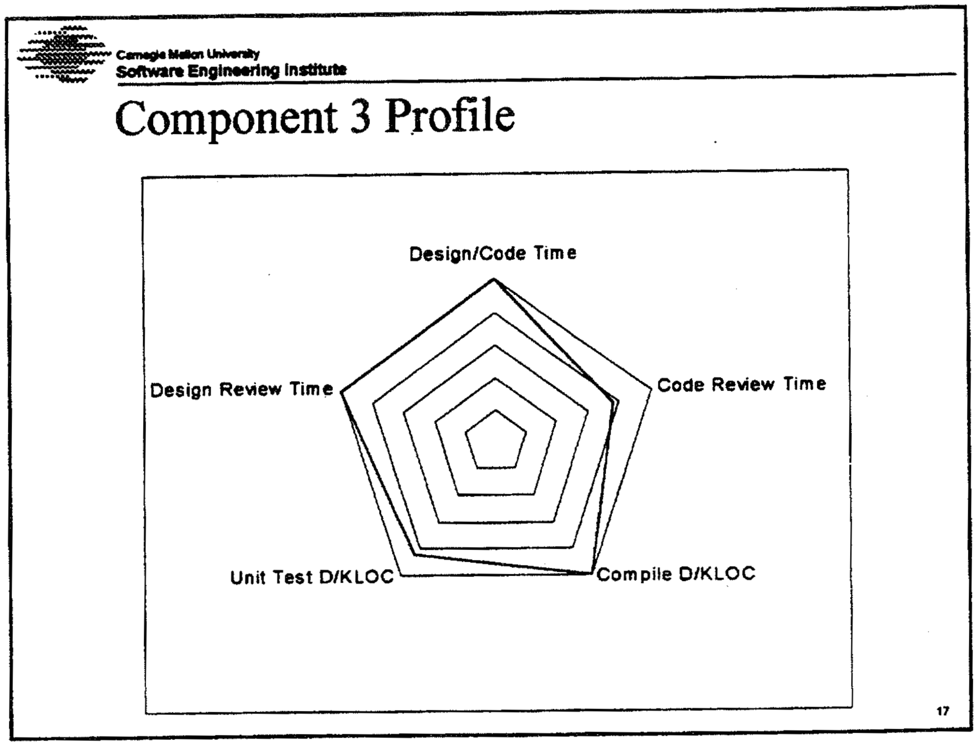
Quality Profiles

The quality profile is a way to judge the likelihood that a product or component has remaining defects.

The profiles are set based on the PSP data.
- **design time > 50% of coding time**
- **design review time > 50 % of design time**
- **code review time > 50% of coding time**
- **compile defects < 10 defects/KLOC**
- **unit test defects < 5 defects/KLOC**

These values are at 1 on the profile with poorer values proportionately closer to the center.

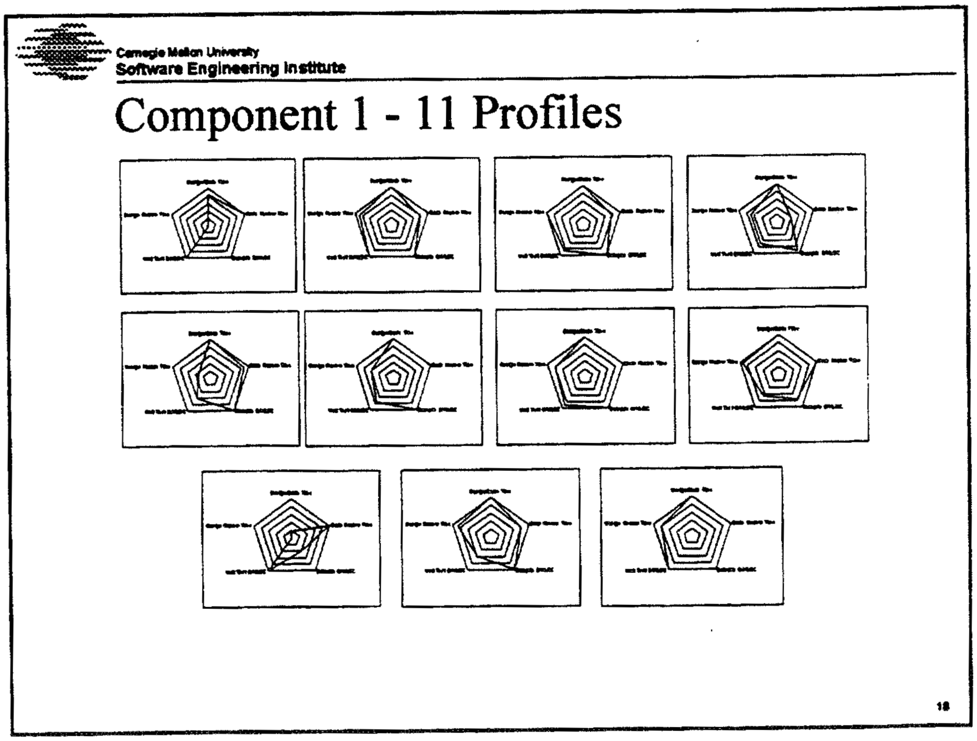
16



Carnegie Mellon University
Software Engineering Institute

Component 3 Profile

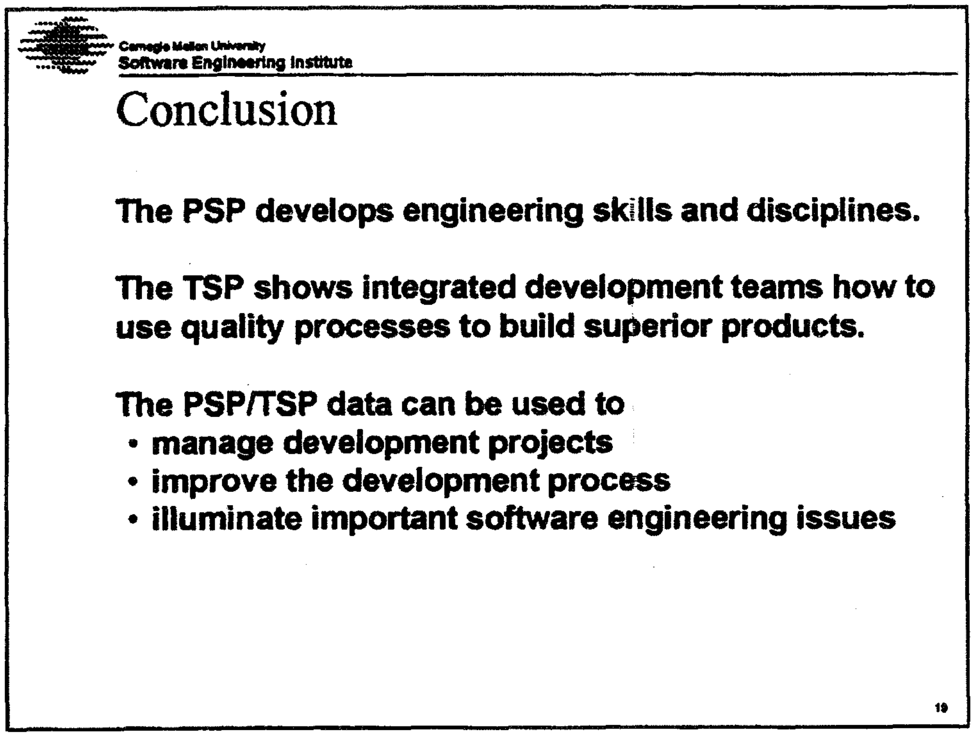
17



Carnegie Mellon University
Software Engineering Institute

Component 1 - 11 Profiles

18



Carnegie Mellon University
Software Engineering Institute

Conclusion

The PSP develops engineering skills and disciplines.

The TSP shows integrated development teams how to use quality processes to build superior products.

The PSP/TSP data can be used to
- manage development projects
- improve the development process
- illuminate important software engineering issues

19

Measuring Impacts of Software Process Maturity in a Production Environment

by

Frank McGarry, Steve Burke, Bill Decker
Computer Sciences Corporation
fmcgarry@csc.com
(301-794-2450)
(Fax- 301-794-8380)

360852

Abstract

For over 20 years, the System Engineering and Analysis Support (SEAS) Center has been supporting NASA in the development of Mission Operations and Data Systems software. During that time, there have been numerous studies and activities designed to measure and analyze the quality of software products as well as the technologies and processes used to produce those products. This paper describes the approach, activities, data, and early results of studies attempting to more generally determine the impact that software processes have on the end item software products. The paper describes the approach to identifying project data for both the product and process and describes one aspect of the series of studies being carried out; the results of analyzing the relationship between project quality and the corresponding process ratings as defined by the Software Engineering Institute (SEI) in their Capability Maturity Model (CMM).

The studies have taken place within CSC/SEAS Center where software systems are developed and maintained in support of NASA flight projects. The projects have ranged in size from 10 KDSI (thousand delivered source instructions) to over 700 KDSI and the applications have specifically been for mission support activities; which include systems for control center operations, data processing activities, command and control and flight dynamics disciplines.

In order to carry out this analysis, over 90 software projects were analyzed where information was available characterizing both the end software product as well as the methods and general processes used to produce that product. Defect data, effort, cycle time, and size were some of the product measures examined. CMM Key Process Area (KPA) ratings from Software Capability Evaluations (SCEs) and Software Process Assessments (SPAs), In-Progress Process Audits (IPPAs), Flight Dynamics Subjective Evaluation Forms (SEFs), and project history reports were some of the measures and information examined representing the process of the software.

This particular study analyzed the potential impact that the CMM Maturity Level 2 and 3 KPAs had on product defect rates, productivity, cycle time, and effort variance. Also analyzed was how these four product measures changed over time. The study showed:

1. There was not a significant correlation between quantified process maturity and the four product measures (for those projects with detailed CMM scores)

2. Software productivity and software defect rates improved consistently over the 14 year period; independent of software process activities based on CMM improvements.
3. Software development cycle time and software effort estimation improved significantly for the SEAS Center after the start of process improvement activities based on CMM (for 1 class of systems).

1. Background and Introduction

Although it is assumed that there are processes which will improve quality and decrease cost for particular applications, there is not common agreement as to which processes are effective and which processes are the most important. This is true even within a single small specific domain. The CMM (Reference 1) defines 18 general attributes of presumably good software development practices whereby the more of these processes that are applied, the less risk there is in having a failed project. There have been several studies carried out in an attempt to verify the assumption that organizations which have improved their maturity of process have improved the quality of their software (References 2, 4, 5). Unfortunately, there is not a large number of such studies and even the ones that exist often rely on an extremely limited amount of empirical evidence which relate the process used to the product generated. Because of the limited amount of available empirical data, subjective surveys are used to gain insight into the effectiveness of improving process maturity (e.g. Reference 3).

1.1 Goals Of The Study Series

Although each project or organization may have their own specific goal or measure of improvement, for this study four general attributes of good software are defined:

1. Productivity (as defined by staff effort per unit of software)
2. Quality (as defined by defect rates)
3. Cycle time (total time required from project start to delivery) normalized by size
4. Effort variance (ability to estimate effort as close to actual effort as possible)

Some of the challenges that previous empirical studies have presented include:

- To what degree can software process be measured consistently ?
- To what degree are successful processes from one production project applicable to another project? If certain processes are successful with a project, will they be successful on another project?
- Can the impacts of specific software processes be quantifiably measured? Which practices are most beneficial in improving the software product?

The goal of the ongoing series of studies at CSC is to determine which process characteristics are beneficial to which class of projects (and which are beneficial to all software projects) By identifying the most appropriate processes (as measured by its impact on the product), the organization will be able to consistently produce more cost effective software and should be able to select those processes which will are most appropriate for specific project goals.

The data used to carry out these studies include detailed project product data (cost, size, defects, cycle time, etc.) and project process data (KPA ratings, Subjective Evaluation Form data, internal audit results using standard processes as benchmarks, and ISO audit results).

1.2 General Approach To Addressing Questions In This Study

The SEAS Center has produced a large number of software systems over the past 10 to 15 years and most of these projects have captured detailed product data and many projects also recorded process information from multiple sources.

This particular study uses the CMM data (including all KPAs for Levels 2 and 3) as the benchmark of process quality. The data for this study was derived from measures which were determined by both independent auditors (SCEs) as well as internal auditors (SPAs). SEAS projects were rigorously audited against the CMM benchmark to derive specific process ratings for the KPAs for multiple projects. Results of CMM audits include an assessment of the organization as a whole (as represented by a sample set of projects) and an assessment of the individual projects which were used to represent the organization. Each of the KPAs is rated as not satisfied, partially satisfied, or satisfied for the organization as a whole and for each of the projects.

Over the period of time where the CMM process rating has been used on SEAS, a series of projects underwent detailed auditing to assign ratings for all 13 KPAs for all sampled projects. This information was used to attempt to make the determination of the relation between higher process maturity ratings and the quality of software product. The analysis looked at 3 aspects of the projects including:

1. For projects with detailed CMM ratings, what is the relation between process maturity and product quality? This analysis looked only at projects which received detailed CMM ratings and also had accurate product data.
2. Independent of process ratings, what is the trend of key product measures over time for all projects developed at the Center; including the 7 years prior to any activity with CMM assessments?
3. Is there any difference between improvement rates prior to CMM activities and after CMM activities?

2. Data and Information

2.1 Product Measures Collected By SEAS

Over 90 SEAS projects are used in these studies as a baseline and for analysis of trends. The sources of the data were standard SEAS data collection forms including Project Closeout Forms (PCFs), Quarterly Summary Forms (QSFs), defect reports, routine contract accounting data, and data obtained directly from project managers.

Table 1 shows a sample of key metrics (estimated and actual effort in staff months, start date, end date, project cycle time in weeks, total errors found in testing, and size in Weighted Delivered Source Instructions (DSI)) for projects involved in this study.

Table 1 - Sample Projects' Product Metrics

Project	Est. Effort	Actual Effort	Start	End	Weeks	Total Errs	WDSI
P1	231.7	324.2	10/1/91	9/16/94	154	190	85900
P2	54.8	55.1	8/1/92	12/15/93	72	16	33511
P3	117.42	593.5	3/1/92	3/1/97	261	109	246950
P4	681.5	560.0	10/1/94	9/30/98	209	298	126500
P5	374	578.1	1/1/88	2/2/93	266	282	84028
P6	N/A	307.0	10/1/95	10/1/97	104	294	146971
P7	677.9	1811.4	10/1/87	8/13/92	254	381	186827
P8	73.1	39.5	1/7/91	4/15/92	66	0	11800
P9	95.2	131.8	6/1/93	5/15/95	102	165	82761
P10	532.9	592.0	11/15/86	9/30/92	307	388	123395
P11	299	422.6	2/10/90	8/15/92	131	585	118698
P12	531.4	559.9	10/1/87	9/22/93	312	470	57774
P13	1865.1	2096.3	10/1/91	9/30/95	209	701	314218
P14	597.6	262.2	2/1/90	2/1/94	N/A	186	52260
P15	1468.9	1562.0	8/1/88	8/6/93	261	123	163938

The definitions of the key measures are given below:

Effort – Staff months spent from project start (start of software specs) to delivery to operations; all roles (managers, developers, testers, QA, CM) and all phases (requirements, design, code, and test) are included with a single significant exception: Approximately half of the projects included in this study did not archive the system engineering and requirements definition effort because this work was done by another organization. The effect of this difference is discussed later.

Size – Measured in delivered source instructions (DSI), that is, non-comment, non-blank source code records.

Weighted size – Total new DSI plus 25 % of total reused DSI

Reuse - Reused code is code that is reused verbatim plus code reused with less than 25% of the code changed.

Defects – Number of errors found by independent testers before delivery that require a change to the executable code; defects do not include unit test or errors in documentation

2.2 Product Measures Used In This Study

This study examines the following four derived measure in detail:

Cycle time – Number of calendar weeks from project start to delivery normalized by size.

Productivity – Weighted DSI per staff month for a project

Effort variance – Difference between actual total effort and estimated total effort (absolute value) divided by estimated total effort.

Defect Rate - The number of defects normalized by weighted KDSI.

2.3 Process Measures Collected By SEAS:

Process information for SEAS projects is available from multiple sources.

- Results of CMM based SPAs and SCEs
- Detailed internal audit results carried out by independent Quality Assurance Office (QAO) and recording process data in the Process Assurance Cycle (PAC)
- Subjective Evaluation Forms (available from many of the Flight Dynamics projects)
- Post development reports which capture process information
- Software Engineering study reports (such as Software Engineering Laboratory reports)

2.4 CMM-Based Process Measures Used In This Study

SEAS has extensive experience with CMM based evaluations. Seven projects were rated against the CMM benchmark during a SEAS-wide CMM SPA in April, 1991. Eleven different projects had independent CMM based software process audits between 1992 and 1994. Four projects were rated against the CMM benchmark in a SEAS-wide SCE in February, 1996. Four projects were rated against a SEAS-wide SCE in November of 1997.

The SPAs and SCEs rated the projects on their compliance with CMM KPAs. The number of activities in a KPA that a project complied with was recorded along with the total number of activities in a KPA. If all activities were complied with, that project was rated fully satisfied in that KPA, if more than half of the activities but less than all of them were complied with, the project was rated partially satisfied. If less than half of the KPA activities were complied with, then a non-satisfied rating was given.

An overall measure of process maturity and conformance ('CMM score') was computed for each project based on CMM SPA or SCE ratings. For each KPA assessed, a score of 3 was given if the project fully satisfied the KPA, 2 was given for partial satisfaction, and 1 was given otherwise. If a project was assessed on 12 KPAs, the project could receive a maximum score of 36. The project's raw point score is scaled against its potential maximum score to a range of 1 to 3. If a project scored 30 from an assessment of 12 KPAs, then its scaled CMM score is 30/12 or 2.5. Although the CMM level ratings are computed differently, the scale range of 1 to 3 was chosen to simulate the 3 CMM levels an organization is typically rated against.

Some projects were only assessed/evaluated on a subset of CMM Level 2 and 3 KPAs. Some early SPAs/SCEs were done using the older (first) version of the CMM - which had a different number of KPAs.

Table 2 shows sample projects with ratings for each project by KPA. The earlier CMM version used in the 1991 SPAs had different definitions for KPAs so that these earlier ratings had to be mapped into the current KPA definitions. The review of this mapping is still ongoing.

Table 2 - CMM SPA and SCE Scores

Project	RM	SPP	PTO	SQA	CM	SSM	OPF	OPD	IC	ISM	TP	SPE	PR	CMM Score
P1	PS	S	S	PS	NS	---	PS	S	---	---	PS	S	PS	2.3
P2	PS	S	PS	PS	PS	---	S	PS	---	---	PS	PS	PS	2.20
P3	PS	S	S	S	S	---	S	S	S	S	S	S	S	2.92
P4	S	S	PS	PS	PS	S	S	PS	PS	PS	S	S	S	2.54
P5	----	S	S	PS	PS	---	S	PS	---	---	S	----	PS	2.50
P6	S	S	S	S	S	---	S	S	S	S	S	S	S	3.00
P7	----	S	S	S	S	---	S	S	---	---	S	----	PS	2.88
P8	----	S	PS	PS	PS	---	S	PS	---	---	PS	----	PS	2.26
P9	S	S	S	NS	S	---	PS	S	---	---	PS	PS	S	2.38
P10	----	S	PS	NS	NS	---	S	PS	---	---	S	----	NS	2.00
P11	----	S	PS	S	PS	---	S	PS	---	---	S	----	PS	2.50
P12	NS	PS	PS	PS	NS	---	PS	NS	---	---	PS	PS	PS	1.70
P13	PS	S	PS	S	S	S	PS	S	S	S	S	S	S	2.76
P14	S	S	S	S	PS	---	PS	PS	---	---	NS	S	S	2.35
P15	S	S	PS	PS	NS	---	NS	S	---	---	PS	S	NS	2.16

2.5 Data Validation and Elimination

The software product measurements as well as the CMM benchmark data went through a data validation process. Some product measurements were suspect as to their validity and therefore those projects were eliminated from the study. Other projects had product measurements but no CMM benchmark data. Figure 1 below shows that out of 95 SEAS projects examined, only 15 had both valid product measurements and CMM benchmark data. Over 70 had valid measurements but no CMM data. The six projects with CMM benchmark data but suspect product measures are not used in this study.

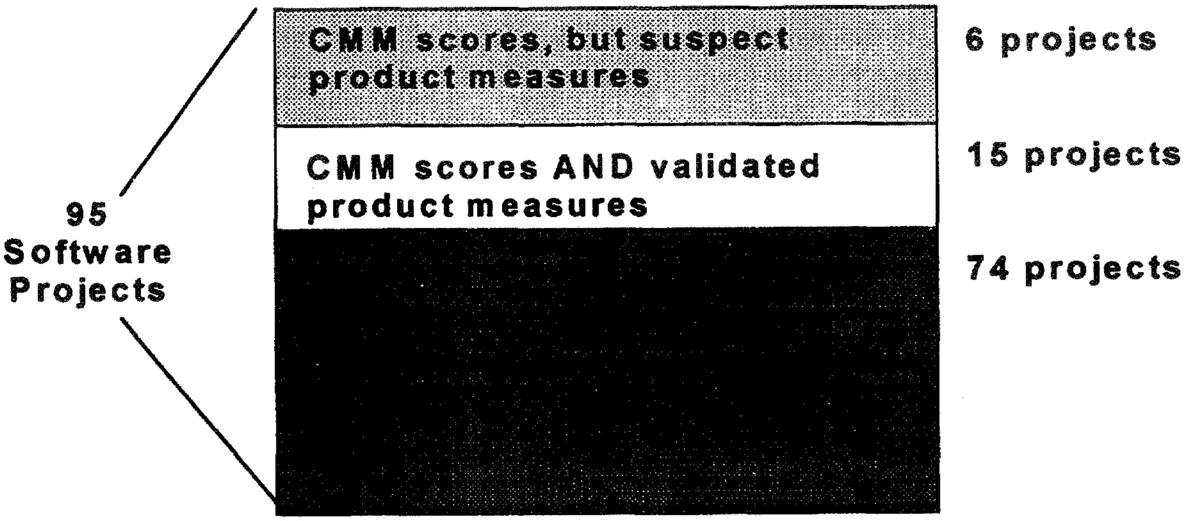

CMM scores, but suspect product measures
6 projects
CMM scores AND validated product measures
15 projects
74 projects
95 Software Projects

Figure 1. Data Validation and Elimination Was Applied Across 95 Projects.

2.6 Assumptions Used In the Study

This series of studies attempts to determine the correlation between process ratings as defined by the SEI or other process measuring scheme and the quality of the end product as measured by the product measures defined earlier. There are numerous factors which impact the outcome of a project including:

- Experience and capability of the personnel
- Problem domain complexity
- Requirements stability
- Environment (and its constraints)
- Available technology and its complexity

The authors realize the numerous factors that influence the quality and success of a software project, but several assumptions are being made for this one particular analysis of how process maturity may impact process.

1. Personnel are random and their capabilities were not the dominant factor in differences.
2. Projects were grouped into 2 domains. This division was employed because projects in Domain 2 effort data did not archive system engineering and requirements definition effort since this work was performed by another organization.
3. It is assumed that the requirements stability was similar for all projects. An analysis of one domain's requirements changes as documented in the project history reports show the number of changes decreasing from the mid 1980s to the present, but there were still a few late requirements changes that caused major system changes in present projects. Therefore, the impact of requirement changes is still relatively high as well as constant (i.e., constantly high).
4. SCE/SPA teams produce consistent results with no significant bias.
5. The application domains and requirements had no significant increase in complexity. A measure of attitude control sub-system complexity for the satellites supported during the second half of the 1980's and the 1990's showed no significant increase in complexity.

3. Analysis

Two approaches were used to analyze the impact of process on products measures. The first performed a direct correlation analysis of the product and process measures. The second approach looked at changes in product measures over the time period immediately before and after the process benchmarking activities began.

3.1 Correlating CMM Score to Product Measures

This approach uses 15 projects for which the most complete process and product information is available. In this approach, no attempt is made to use the information which rated the organization as a whole, only the specific project data was used. Correlations (r^2) for CMM score versus each of the product measures were derived and are shown in Table 3. Figure 2 shows one example of the scatter plots, CMM score versus defect rate.

Table 3 - Process vs. Product Measure Correlations

CMM score vs.	r^2
Defect rate	0.19
Productivity	0.25
Cycle time	0.50
Effort Variance	0.15

While the correlation analysis indicated the expected trends (e.g , higher CMM score associated lower defect rate) none of the correlations are significant.

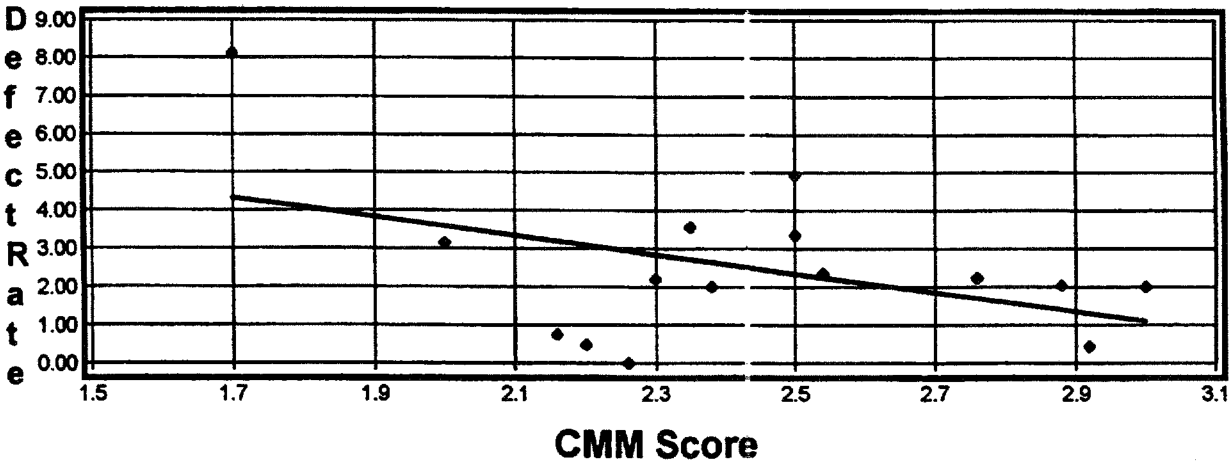


Figure 2. Defect Rate Versus CMM Sccre for 15 Projects.

3.2 Effect Of Process Benchmarking Over Time

The second approach grouped all 89 projects with valid product data into two application domains. A yearly average of all projects active in that year was derived for the four measures of productivity, defect rate, cycle time, and effort variance. Each series of yearly averages reflects the product performance of the organization as a whole over the dimension of time. The time ranged from 1984 to the present (1997). CMM benchmarking improvement activities were not started until 1991. Therefore, we have a good sample set of about six to seven years prior to the benchmarking activities that can serve as a baseline.

3.2.1 Product Measure Trends Over Time

The trends for productivity and quality show a steady relatively linear improvement across the whole 13 to 14 year time span (the example in Figure 3 shows defect rate). The correlations are very good for this set of data with values greater than 0.6 for productivity and defect rate for both domains (Table 4).

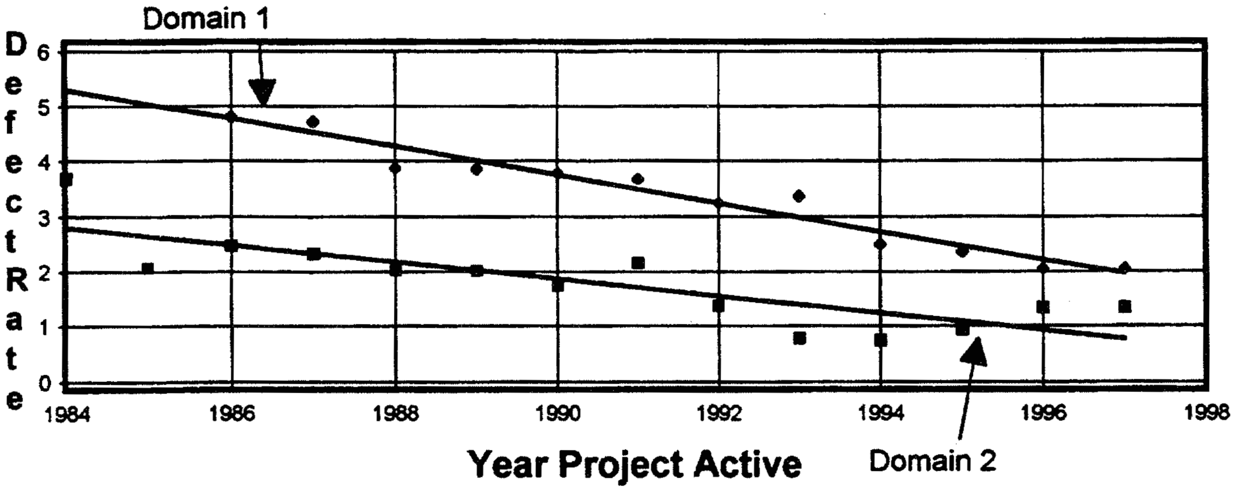


Figure 3. Yearly Average Defect Rate for Domains 1 and 2

Table 4 - Time Vs. Product Measures Correlations and Yearly Improvements

Time vs.	r^2	Avg. Yearly Percent Improvement	r^2	Avg. Yearly Percent Improvement
	Domain 1		Domain 2	
Defect Rate	0.95	5.3%	0.68	5.6%
Productivity	0.72	6.7%	0.87	5.8%
Cycle Time	0.30	-	0.04	-
Effort Variance	0.74	5.9%	0.60	-14.7%

The trends for cycle time and effort variance had mixed results over the full time span and across Domains. The cycle time trend had very weak correlations across the full time span for both domains. The effort variance trend had good correlations for both domains. However, Domain 2's trend showed a retrogression in estimation ability. That is , the variation was increasing about 15% per year as shown in Figure 4. It is currently not clear what the real cause of this retrogression is.

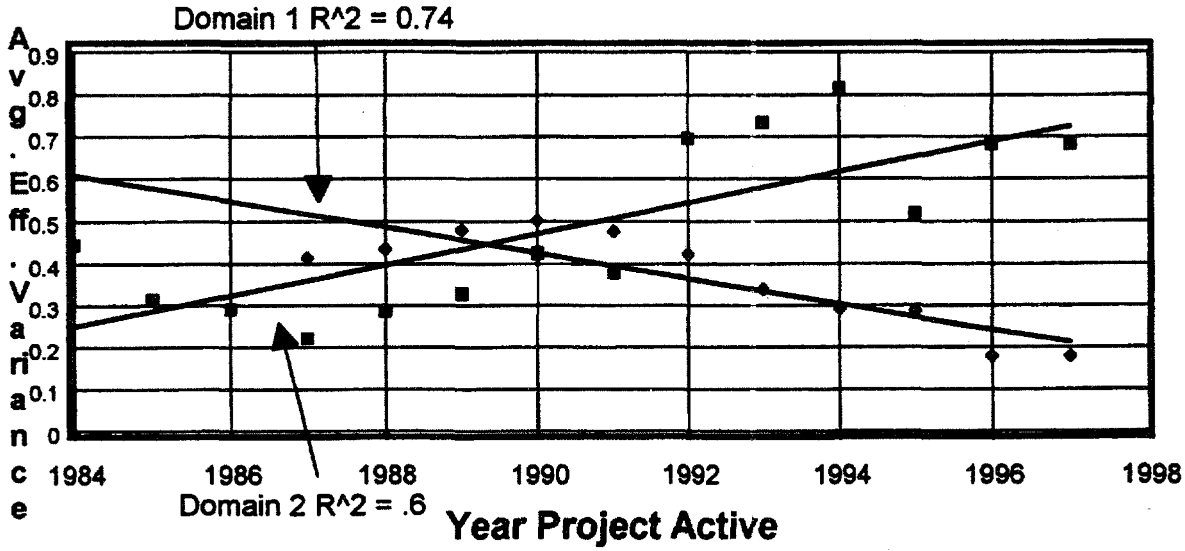


Figure 4. Yearly Average Effort Variance for Both Domains.

3.2.2 Impact of Benchmarking Activities on Product Measures Over Time

The final analysis presented in this study analyzed two time periods. The 1984 to 1990 time period was compared to the 1991 to 1997 time period. The CSC/SEAS Center began CMM benchmarking improvement activities in 1991 and has continued them up to the present. A noticeable change in the slopes across the two time periods could be interpreted as a benefit resulting from benchmarking improvement activities.

As shown in Figure 5 and Table 5 below, there is no significant change in slope across the two time periods for productivity and defect rate.

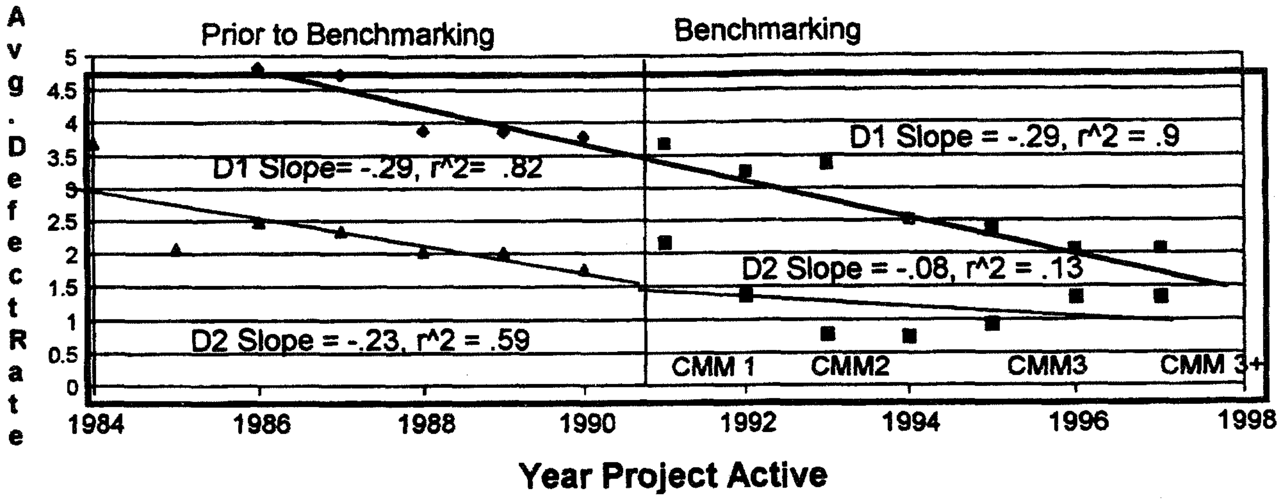


Figure 5. Yearly Average Defect Rate in Two Time Periods.

Table 5. Slopes and Correlations for Yearly Average Product Measures Across Two Time Periods.

	Pre-Benchmarking		**Benchmarking**	
	Slope	**Corr.**	**Slope**	**Corr.**
D1 Defect Rate	-0.29	0.82	-0.29	0.90
D2 Defect Rate	-0.23	0.59	-0.08	0.13
D1 Productivity	33.4	0.70	37.0	0.79
D2 Productivity	18.0	0.72	14.3	0.54

For cycle time and effort variance, however, Domain 1 did show a positive turnaround (i.e., improvement) during the benchmarking improvement activity time period. Figure 6 shows a complete change in the sign of the slope of the rate of change of cycle time for Domain 1. Domain 2 showed no discernible change. Figure 7 shows a similar reversal for the Domain 2 effort variance product measurement.

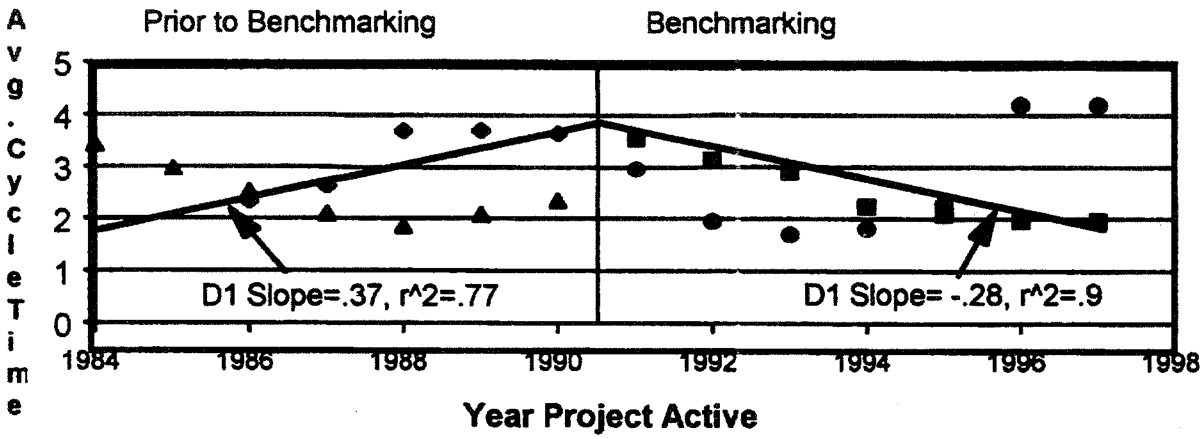


Figure 6. Changing Trend of Domain 1 Cycle Time.

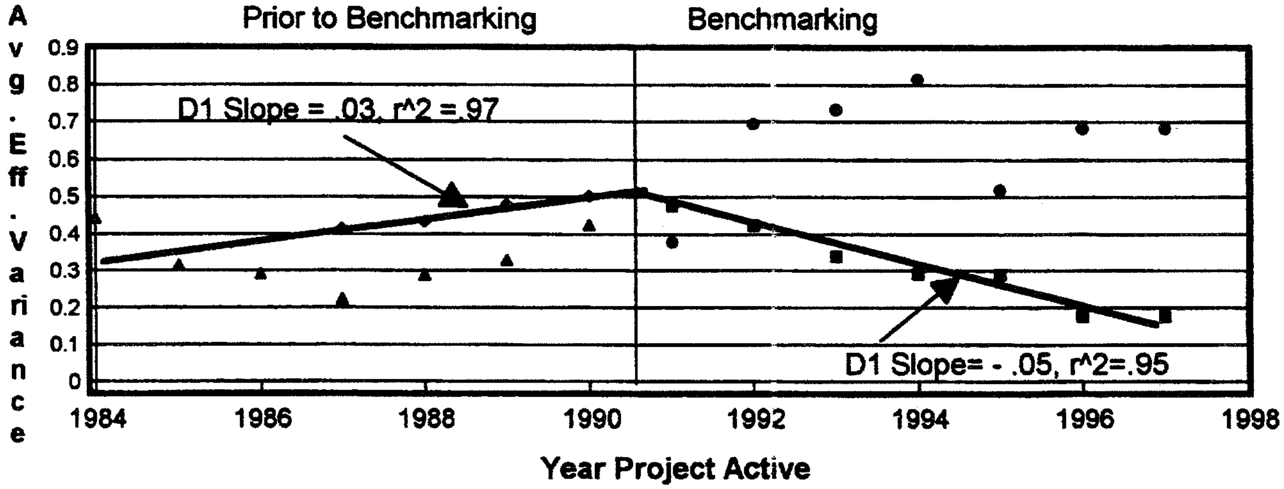


Figure 7. Changing Trend of Domain 1 Effort Variance.

4. Results and Implications

A conscious attempt was made to gather valid empirical data to determine the quantifiable impacts process improvement benchmarking activities have on product measures. An analysis was performed on a set of data with a sufficiently long prior set of baseline data in an attempt to filter out the impact time by itself may have on product measures. Improvement in productivity and defect rate has been consistent for the past 14 years (1984–1997). The benchmark ng activities during the second half of the 14 year period did not have a perceptible impact on these product measures. The summary results from this single study are:

- Productivity increased at a consistent rate over the 14 years in which valid data was available. The rate of increase was approximately 6% per year.

- Defect rate decreased at a consistent rate of 5% per year for all 14 years.
- Compared to the time prior to the start of activities, no significant change in improvement rates were noticed after "process benchmarking" activities started.
- Significant changes were noticed **in one domain** for cycle time and effort variance after benchmarking activities started.
- Benchmarking activities showed a negative relation with effort variance **in one domain.**
- Combining large numbers of projects for study could be misleading unless specific domain characteristics are known.

The observed minimal impact of process maturity on product measures could have alternative explanations such as:

- The measure of process maturity used by this study (a composite of numeric values for degree of satisfaction of each KPA) was not valid or accurate.
- The sample set of only 15 projects with both process and product measures was too small and/or not representative of the organization.
- CMM ratings may be more indicative of 'organization' maturity as opposed to 'project' maturity as used in this study.
- SCE teams have matured significantly over time and produce more accurate and consistent ratings in 1997 than in 1991.
- Changes in the complexity of projects directly affected the product measures.
- The study period was too long since the CMM approach has changed between 1991 and 1997.
- Technology changes have overwhelmed the impacts inherent in process changes.

This one study has only taken a single narrow view of analyzing the impacts of software process characteristics on the end software product. There certainly is no evidence that increased maturity ratings do not favorably impact the software product; but there is evidence that the analysis of software process requires many additional empirical studies and a much more thorough analysis.

There are several additional implications that the SEAS Center is pursuing:

1. The maturity rating offered by the CMM's 18 Key Process Areas (KPAs) may be incomplete, difficult to quantify, or possibly misleading. Additional process measurement schemes must be analyzed.
2. Without accurate product measures (such as productivity, defect rates, cycle times) being captured along with process characteristics, it will be extremely difficult to verify the value and accuracy of such process benchmarks as the CMM.
3. Without such studies as SEAS is pursuing, process benchmarks (such as the CMM) cannot be self correcting or self improving. There must be such empirical analysis to justify enhancement or change to evolving process benchmarks.
4. Measuring quality (or maturity) of software processes is extremely difficult; in fact, capturing process characteristics quantifiably may be a much more immature science than is commonly assumed.

5. Future Work

The SEAS Center has accumulated a significant amount of software process data and associated product data; and the Center is continuing to collect this information on new and active projects. The long term plan is to effectively implement improvement activities through the selective adoption of the most appropriate set of processes and technologies. Only by measuring and evaluating the effects that various process characteristics have on the end product, can we better understand the most appropriate changes to implement in the specific environment. Additionally, through the continual measurement and evaluation of process characteristics some determination can be made as to which processes are more generically applicable and which are domain specific; this insight will benefit a broader spectrum of development within the software engineering community.

Several parallel studies are currently active within the SEAS Center and several additional ones are planned:

- Analyze additional process measures (other than CMM benchmarking data) to determine impacts on product.
- Analyze CMM benchmarking data to determine consistency of process rankings.
- Investigate product measure impacts of individual process characteristics. (e.g., Do we get more payback from training or configuration management?)
- Determine if ISO 9001 benchmarking activities are quantifiable and usable as process measures.
- Refine monitoring of process and product data on all projects; refine mechanisms for ensuring validity.
- Compare with larger domains to determine consistency of findings in other CSC domains.

References:

1. M. Paulk, et al., "Capability Maturity Model Version 1.1" , IEEE Software, July 1993.

2. J.G. Brodman and D. Johnson, " Return on Investment from Software Process Improvement as Measured by U.S. Industry," Crosstalk, Apr. 1996 (pp. 23-28).

3. J.D. Herbsleb and D.R. Goldenson, "A Systematic Survey of CMM Experience and Results," Proceedings ICSE 18, IEEE Computer Society Press, Los Alamitos, Calif., 1996.

4. P. Lawlis, R. Flowe, and J. Thordahl, " A Correlation Study of the CMM and Software Development Performance," Crosstalk, Sept., 1995.

5. M. Diaz, J. Sligo, " How Software Process Improvement Helped Motorola", IEEE Software, September 1997.

Measuring Impacts of Software Process Maturity in a Production Environment

December 3, 1997

Frank McGarry
Steve Burke
Bill Decker

Computer Sciences Corporation
SEAS Center

Common Concepts of Software Process

- **Process used to develop software will significantly influence software product**
  - Inspections will result in lower defect rates
  - Controlled standards will result in less rework
- **There is a set of Good, Better, and Best Process characteristics** (at least a set of defined benchmarks representing mature processes)
  - CMM (Levels 2 through 5)
  - ISO 9000
- **If an organization identifies and adopts appropriate processes, it becomes a more mature organization as measured by benchmarks such as CMM.**
- **Likelihood that an organization will produce better, more reliable software is higher as maturity of its process increases**
  - Increase productivity
  - Decrease defect rates

Goals of This Study

- **Determine trends over time of key product measures** (Productivity, defect rate, cycle time, predictability variance)
  - Will defect rates decrease over time even when my process maturity does not change?
- **Determine impacts that process changes (improvements as measured by independent benchmarks) have on software product**
  - Does the organization produce better software when it's maturity increases?
- **Identify which process characteristics are most important to organization to drive product improvement**
  - Is training more important than CM?

Study Environment (CSC/SEAS Center)

- **Approximately 1000 professionals supporting NASA**
  - 35 to 50 percent software development and maintenance
- **Projects**
  - Size 5,000 SLOC to 1,000,000 SLOC (typically 100,000 SLOC)
  - Effort 1 sy to 75 sy (typically 25 to 35 sy)
  - Third-generation language (C, C++, Ada, FORTRAN)
- **Application**
  - NASA mission support
    - » Command and control
    - » Data processing
    - » Flight dynamics
    - » Simulation and modeling
- **Process (CSC view):**
  - Even before 1990, SEAS had a good software process in place
    - » Detailed written standards (SSDM)
    - » Consistent, controlled process existed

Source of Data

- **Data was accumulated from 1984 through 1997**
- **Approximately 90 projects have consistent, relatively complete data** (projects with questionable data were discarded)
- **All data was collected during project performance (no data produced after the fact)**
  - Product data included size(new, reused, COTS), dates, language, phases, effort, defects, (as well as other product parameters not used for this study)
- **Process information derived from detailed SCEs and SPAs**
  - SEAS underwent detailed independent SCEs in 1991, 1996, 1997, (Jan. '98)
  - SEAS underwent independent SPAs in 1991, 1992 (4), 1993 (5), 1994(7)
  - 15 of 90 projects were involved in detailed process analysis (samplings from other projects periodically took place during the SCEs and SPAs)
- **All data is reviewed, QA'd, and verified as it is provided to the independent measurement group**
  - Consistent set of definitions and counting processes provided to project personnel (e.g., line of code)

Definitions for Product Information*

- **Size** – Amount of delivered source instructions (DSI). Categories are new DSI and reused DSI (COTS is also tracked)
- **Weighted DSI** – Total new DSI plus 25 % of total reused DSI
- **Effort** – Staff months spent from project start (start of s/w specs) to delivery to operations; all activities (managers, developers, testers, QA, CM) and all phases (requirements, design, code, and test) included.
- **Defects** – Number of errors found by independent testers before delivery that require a change to the executable code; defects do not include unit test or errors in documentation
- **Productivity** – Weighted DSI per staff month for a project
- **Cycle time** – Number of calendar weeks from project start to delivery (normalized by size)
- **Effort variance** – difference between actual total effort and estimated total effort (absolute value) divided by estimated total effort

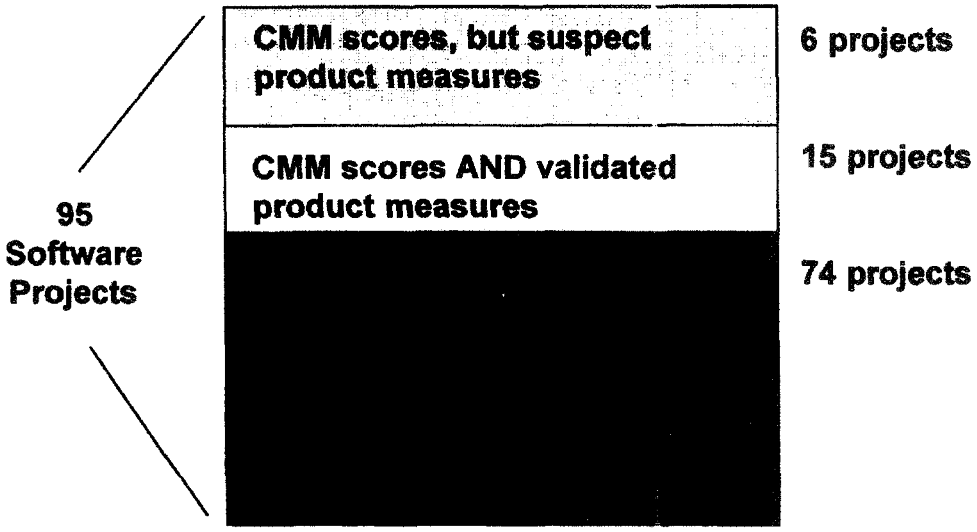
*Definitions consistently applied to all projects

Definitions for Process Information

- **CMM score (for project)** – Score that quantifies compliance of a project to CMM level 2 and 3 key process areas (13 KPAs)
  - Based on SPA and SCE results
  - Projects assessed and evaluated as satisfied, partially satisfied, or not satisfied for each KPA were given a point score of 1, 0.5, or 0
  - Totals for all KPAs then scaled from 1 to 3
- **Additional process measures were captured**, (but are not discussed in this report)
  - Derived scores using a scale of approximately 35 characteristics, where ratings were derived jointly between manager and process group
  - Internal audit process scores – ratings determined periodically during project development based on adherence to predefined set of process characteristics (independent audit)

Data Selection and Elimination

15 of 95 projects had both CMM and product measures


95 Software Projects
CMM scores, but suspect product measures
6 projects
CMM scores AND validated product measures
15 projects
74 projects

Data Selection and Elimination

Data was Divided into Two Domains:

Domain 1 and Domain 2

- **Historical differences in measurement program**
  - Domain 2 had local measurement program long before all SEAS
  - Effort counted differently
    - » Systems engineering included in Domain 1, not in Domain 2
    - » Specs development included in Domain 1, not in Domain 2
- **Problem complexity**
  - Domain 2 very repetitive systems – algorithm based; Domain 1 more one-of-a-kind (unique) full system
  - Domain 2 very stable, unchanging hardware configuration; Domain 1 typically had unique hardware as part of system

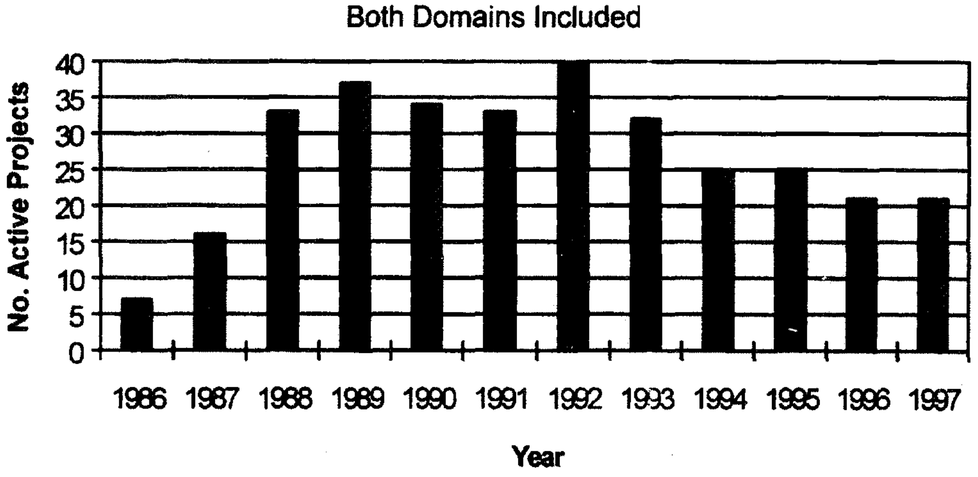
- Domain 1 – 47 projects (with valid product information)
- Domain 2 – 42 projects (with valid product information)

Example of Data Used in Study

Project	Domain	Start	End	Est Effort	Actual Effort	WDSI	Errors	CMM Score	Rating
P1	1	11/15/86	9/30/92	532.9	592.0	123,395	388	2.00	SPA
P2	1	10/1/87	9/22/93	531.4	559.9	57,774	470	1.70	SPA
P3	1	8/1/88	8/6/93	1468.9	1562.0	163,938	123	2.16	SPA
P4	2	10/1/88	10/31/90	109.7	151.1	213,918	83	-	–
P5	2	2/10/90	8/15/92	299.0	422.6	118,698	585	2.50	SPA
P6	2	6/1/93	5/15/95	95.2	131.8	82,761	165	2.38	SPA
P7	1	10/1/91	9/30/95	1865.1	2096.3	314,218	701	2.76	SCE
P8	2	3/1/92	3/1/97	117.4	593.5	246,950	109	2.92	SCE
P9	1	4/1/92	8/30/95	238.5	278	98,000	89	-	–
P10	1	10/1/95	10/1/97	–	307.0	146,971	294	3.00	SCE

- 90 projects covered the period 1984–1997
- Many projects were active 3 to 5 years
- Process ratings determined consistently for all domains
- Some differences in product measures of two domains

Projects Active per Year



3 Views of Project Information

1. Determine correlations of CMM rating and 4 product measures*
   - Productivity
   - Defect Rate
   - Effort Variance (estimation quality)
   - Development Cycle time

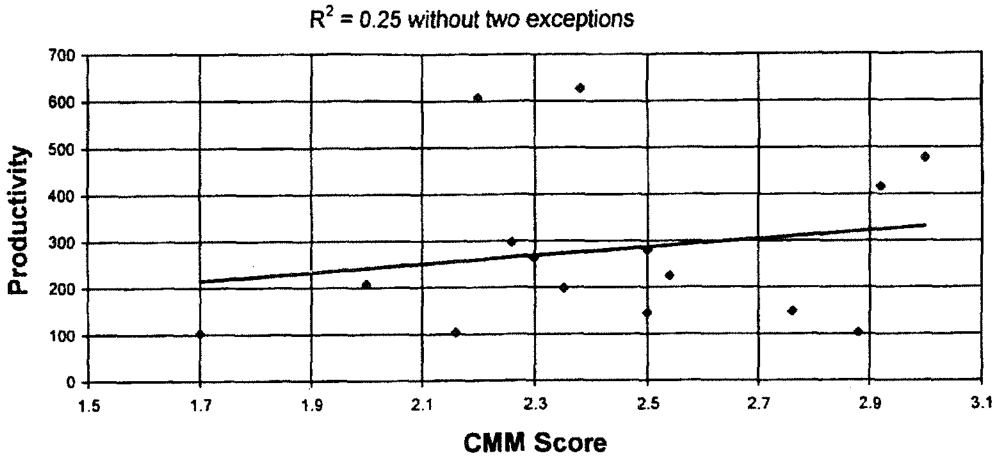
2. Determine trends of 4 product measures over the entire 14 year period (indeperdent of process ratings)

3. Determine trends of 4 product measures before benchmarking activities vs. trends after benchmarking activities

* 15 projects had detailed KPA ratings and valid product data

CMM Score Versus Productivity

(For 15 projects with detailed CMM scores over 7 years)

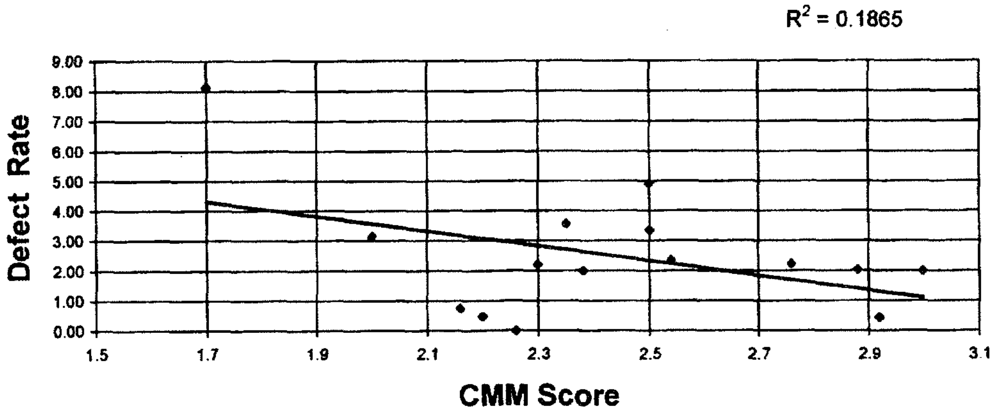


No strong correlation between Productivity and CMM score

- The two domains have very similar *process* score fits
- No significance lost when combined as single domain (for *process*)

CMM Score Versus Defect Rate

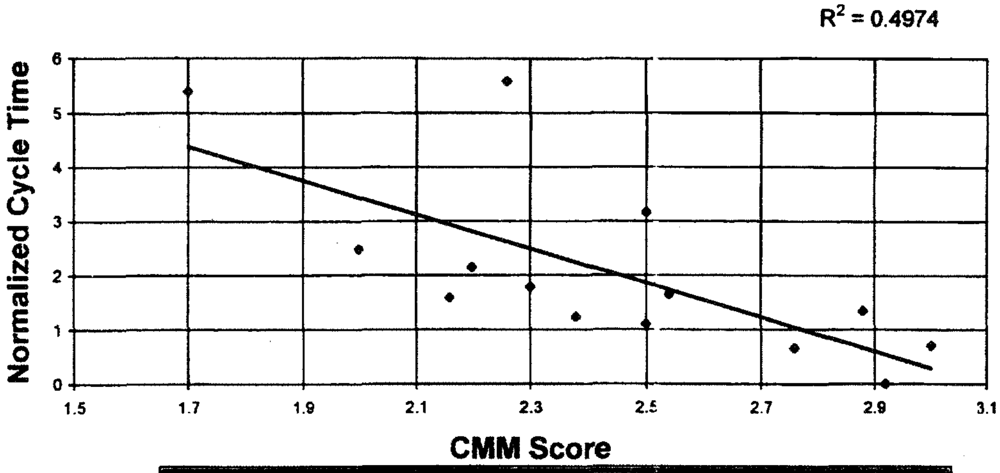
(For 15 projects with detailed CMM scores over 7 years)



No strong correlation between defects and CMM Score

CMM Score Versus Normalized Cycle Time

(For 15 projects with detailed CMM scores over 7 years)

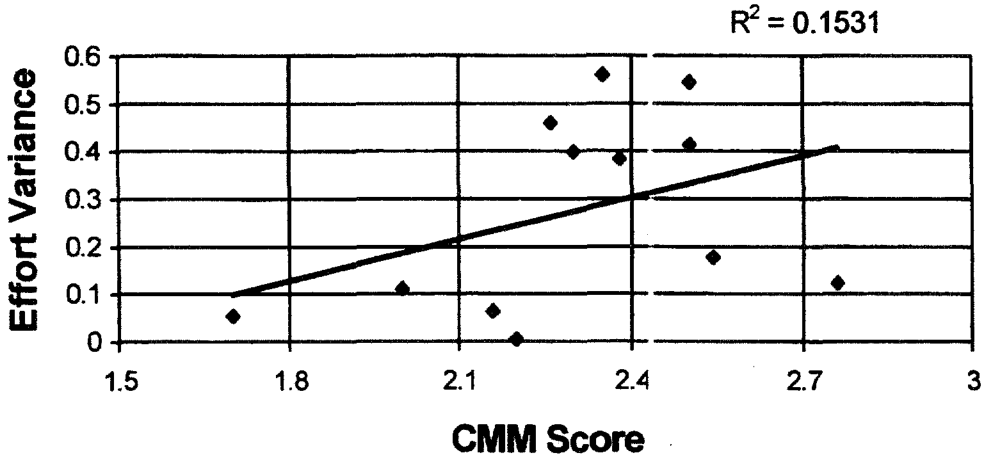


Fair correlation between cycle time and CMM score

(significant to .008 level)

CMM Score Versus Effort Variance

(For 15 projects with detailed CMM scores over 7 years)

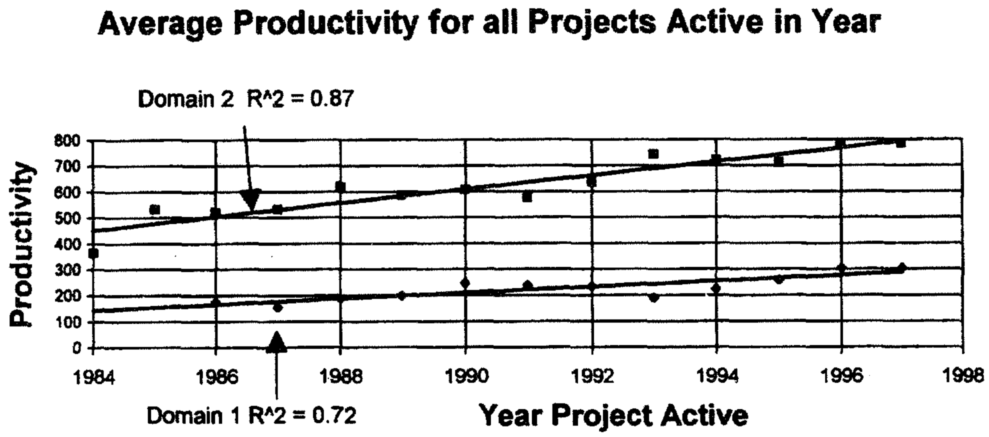


No strong correlation between Effort Variance and CMM Score

2 Outliers removed

Productivity Versus Time

(For all 90 projects over 14 years)

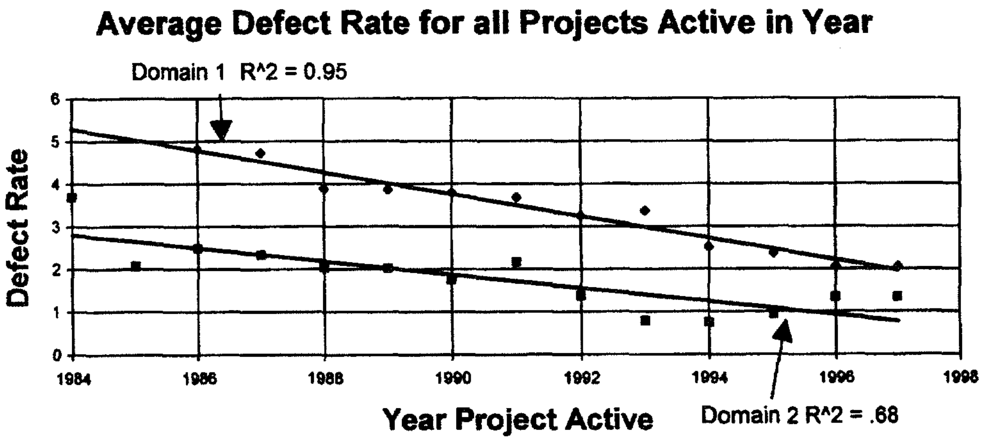


Very good correlations between time and productivity (significant to .003 level)

- Domain 1 improved 6.7 percent per year
- Domain 2 improved 5.8 percent per year

Defect Rate Versus Time

(For all 90 projects covering 14 years)

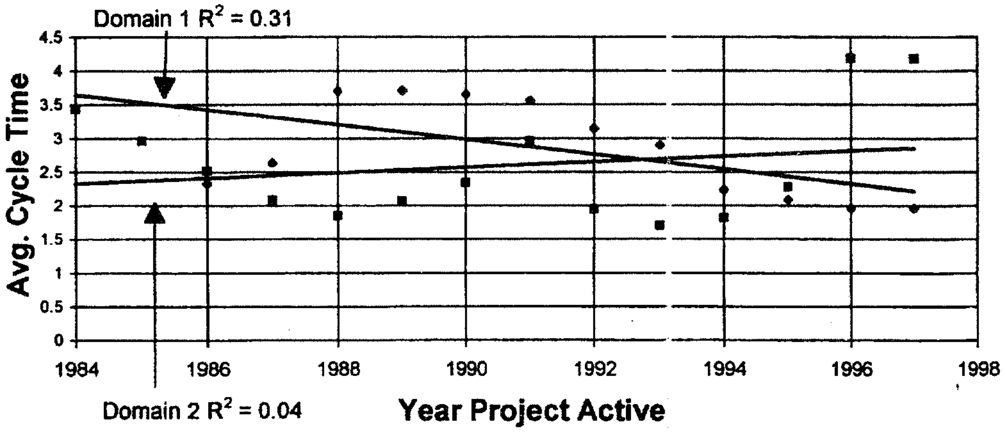


Very good correlations between time and defect rate (significant to .0001 level)

- Domain 1 improved 5.3 percent per year
- Domain 2 improved 5.6 percent per year

Cycle Time Versus Time

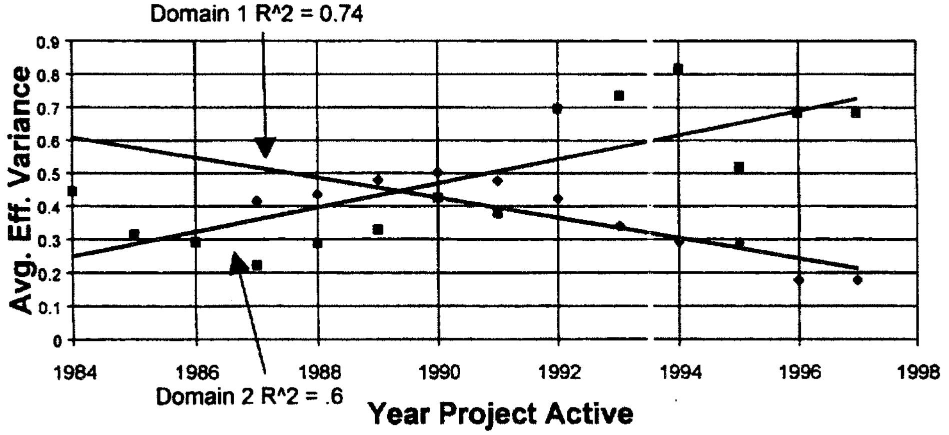
(For all 90 projects covering 14 years)


Average Cycle Time for all Projects Active in Year

Very weak fit for time(14 years) vs. development cycle time

Effort Variance Versus Time

(For all 90 projects covering 14 years)

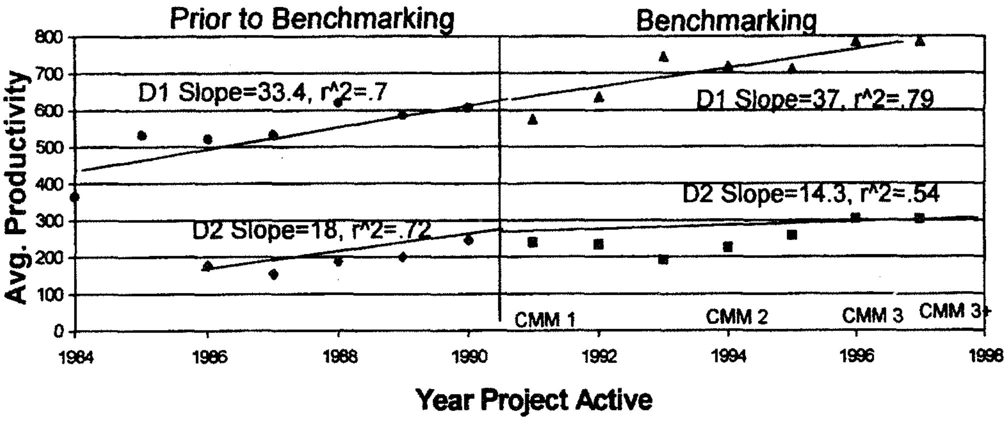

Average Effort Variance- all Projects Active in Year

One domain improved while the other domain got worse over time (14 year fit)

- Domain 1 improved 5.9 percent per year
- Domain 2 regressed 14.7 percent per year

Comparing Productivity Before and After Process Benchmarking Activities*

Average Productivity for all Projects Active in Year

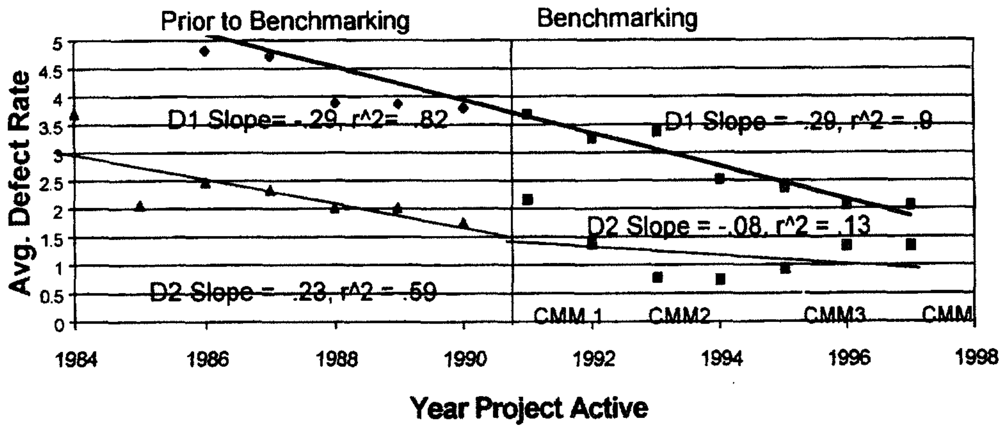


There is no significant difference in rate of productivity change before and after the start of 'Benchmarking' activities

* SCE and SPA activity started in 1991

Comparing Defect Rate Before and After Process Benchmarking Activities*

Average Defect Rate for all Projects Active in Year

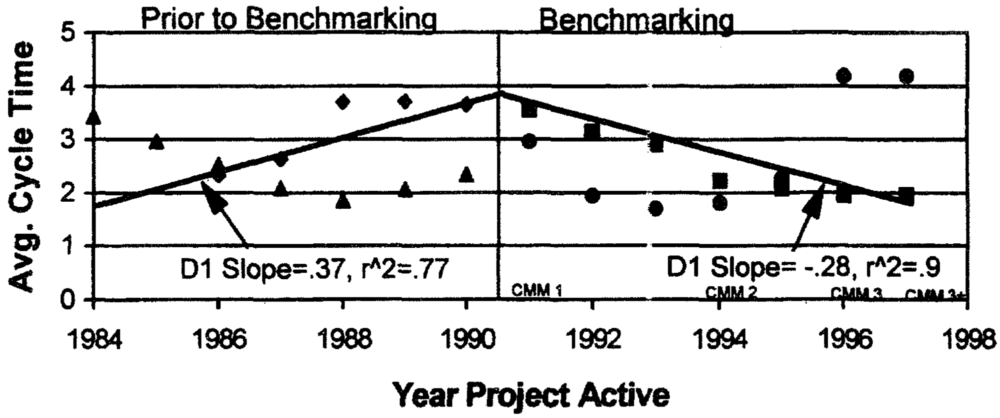


There is no significant difference in the improvement of defect rates before and after the start of benchmarking activities

* SCE and SPA activity started in 1991

Comparing Cycle Time Before and After Process Benchmarking Activities

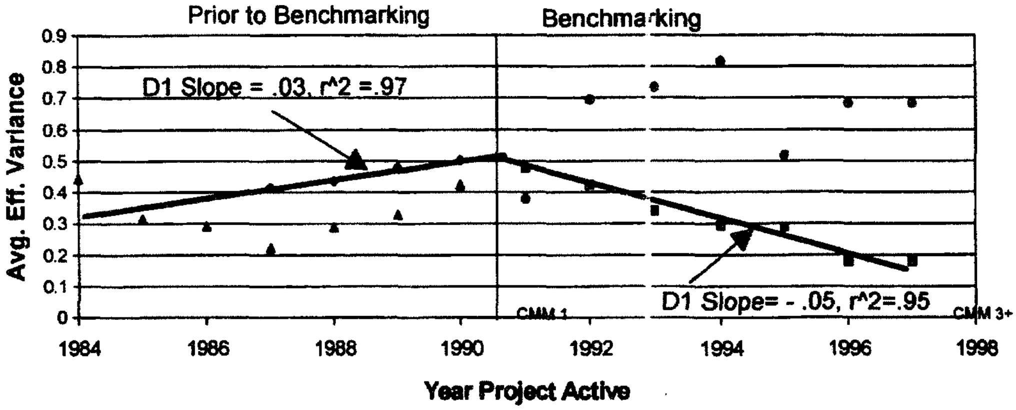
Average Cycle Time for all Projects Active in Year



One domain (D1) shows significant change in cycle time improvement after start of benchmarking activities
(other domain shows minimal improvement)

Comparing Effort Variance Before and After Process Benchmarking Activities

Average Effort Variance for all Projects Active in Year



One domain (D1) shows significant change in Effort Variance improvement after start of benchmarking activities
(other domain worsened)

Summary of the Analysis

- Significant **assumptions** were made in analyzing impacts of process
  - Review teams (SCE/SPA) produce consistent results (no significant bias)
  - Similar complexity of projects for evaluations
  - Same process characteristics are valid for all projects in study domain
- Improvement in productivity and defect rate has been consistent for past 14 years (1984–1997)- Benchmarking activities did not help
  - Productivity increased at rate over 6% per year
  - Defect rate decreased at rate over 5 % per year
  - No change in improvement rates after "process benchmarking" activities started
- Process improvement ratings (CMM) did show a positive relation **in one domain** with cycle time and with effort variance
- Process improvement ratings (CMM) showed a negative relation with effort variance **in one domain**
- Combining large amounts of projects for study could be misleading unless specific domain characteristics are known (Domain 1 Vs. Domain 2)

Questions and Possible Explanations

- Process rating was not valid
  - SCE teams have matured significantly over time (more accurate in 1997 than in 1991)
  - Projects assessed were not representative of the organization
  - Timeframe of study was too short; will have to look at long-range impacts
- Complexity of projects resulted in different product measures
- Time of study was too long – CMM approach has changed between 1991 and 1997
- Technology changes have overwhelmed impacts inherent because of process changes
- Looked at wrong measures

Ongoing and Planned Work

- Analyze additional process measures (other than KPAs) to determine impacts on product; additionally, determine consistency of process rankings
- Continue efforts to measure impacts of individual process characteristics (e.g., do we get most payback from training or conducting empirical studies?)
- Determine if ISO ratings are measurable (use the ISO ratings as process measures)
- Refine monitoring of process and product data on all projects; refine mechanisms for ensuring validity
- Compare with larger domains to determine consistency of findings in other CSC domains

A Time-Based Management Approach to Software Process Improvement

360854

C. L. Beard
P. C. Paesano
C. R. Savage

Abstract

Keywords: Time Based Management, Cycle-Time Reduction, Just-In-Time, Software Process

The software Capability Maturity Model is built upon the framework of Total Quality Management (TQM). In many cases, the application of TQM principles results in cycle-time reduction as a side benefit of the process and quality improvements.

TQM originated in manufacturing, and the evolution of TQM in the manufacturing world has been Time-Based Management (TBM). TBM deals with a change of perspective, from a focus on the quality of products to a focus on cycle time reduction, while still maintaining high standards of quality. TBM provides a very straightforward metric, cycle time, which allows one to measure the process from a different perspective. By focusing on cycle-time reduction, while still maintaining a TQM foundation, signification improvements in the process can be realized. Also, it has been demonstrated that cycle-time reduction is a important factor in achieving a competitive advantage.

This paper demonstrates how we took the models for TBM in the manufacturing area and adapted them for use in a white collar software "factory". It explains the basic concepts, ties them into the Capability Maturity Model, and provides some feedback on our experience.

Introduction

The Capability Maturity Model (CMM) was developed by the Software Engineering Institute (SEI) to define levels of software process maturity and to identify key process areas which focus on increasing the maturity level of a software development organization.[8,9] The CMM was modeled on the Total Quality Management (TQM) concept, which has its roots in manufacturing facilities.

Using TQM as the foundation, processes in manufacturing organizations have since evolved to a new level in continuous process improvement, Time-Based Management (TBM). **It is important to note that a TBM focus does not supplant the quality focus; instead it requires the total commitment to quality as the foundation upon which the time-based management system is built.**

Time-Based Management is also referred to as Cycle-Time Reduction (CTR) or Just-In-Time (JIT). These all focus on cycle time as the fundamental metric to assess a project. Cycle time is defined as the time from the inception of a project until the completion of the project in real calendar time. This is a direct component of other measures such as product introduction cycle time, time to market, and defect fix turn-around time.

Knowing that "time is money", the TBM metrics all have a direct business impact. The time to produce a new product or to keep ahead of competitors' offerings is of extreme competitive importance in the volatile software market. In addition, there is the cost of lost opportunities which result from not having a product available in time for the market demand. Even in those areas where competitiveness is not a direct issue, such as in government contracts or internal development, cycle-time is a very important attribute from a business perspective. Focusing on cycle-time reduction leads to a significant improvement in product quality while costs decrease by reducing overall development and maintenance effort.

Background

TBM was first documented at Toyota for the development and manufacture of automobiles[2,7,10]. Toyota reduced the time to develop a new automobile model from the industry average of over five years to under three years, and since then has reduced the time to under two years. At the same time, the quality of the cars manufactured by Toyota and the manpower effort to manufacture a car continued to improve. These process changes were copied by other Japanese automobile manufacturers and gave the Japanese a significant competitive advantage. It was only after the US automobile manufacturers started copying these practices that the competitiveness of the US manufacturers improved. Chrysler has introduced a large number of new models on the market, primarily because it reduced the cycle time down to under three years while simultaneously reducing the cost[10]. It now has a development cost for a new model which is significantly less than General Motors, and not surprisingly, has a much larger profit margin per unit sold.

The Westinghouse Commercial Nuclear Fuel Division (CNFD) is primarily a manufacturing organization of a product (commercial nuclear fuel) which requires a very high degree of reliability. Implementation of the product requires a significant amount of complex engineering analysis to demonstrate appropriate margins to safety. As such, we have had a very high level of quality focus within the organization for over 25 years, and were one of the first winners of the Malcom Baldridge National Quality Award. It is the responsibility of the software development organization in CNFD to develop and maintain the software for the engineering analysis of the product and its use. The software is also sold to our customers to enable them to perform their own independent analyses.

Our entire organization (both manufacturing and office support) has focused on Time-Based Management. As such, we have taken the principles of TBM and applied them directly to our software process. Many individuals in our organization are involved in our local SEI sponsored Software Process Improvement Network (SPIN) chapter, and we have used the CMM as a basis for our process improvements. However, by emphasizing our focus on cycle-time reduction and using the concepts of TBM, significant changes to the software process have been made which have demonstrated substantial benefits to our software organization.

At the 1996 Software Engineering Process Group Conference, a paper was presented on *Software Cycle-Time Reduction* by Herb Krasner[3]. In that paper, evidence was presented that cycle-time reductions were occurring as an organization improved on its process maturity. However, the paper pointed out that there was very little information in the literature about applying the practices of cycle-time reduction to software.

Recently, there has also been a number of books published by Microsoft Press describing the Microsoft process[1,4,5]. Although most of these do not directly focus on cycle-time reduction, many of the key practices which are described are very similar to some of the practices which we evolved out of our cycle-time reduction program.

Thus, we have taken the CMM and prioritized process changes from the perspective of cycle-time. We have also taken good, solid ideas from other sources and have adapted them to the software development process, resulting in significant software process changes. Many of these changes are not obvious without looking at the software engineering process from the perspective of cycle-time.

Key Elements of Just-In-Time

There are eight key elements of Just-In-Time processes:

- Continuous Flow
- Multi-functional Workers
- Pull Through System
- Level Production
- Set-up Time
- Product Quality
- Machine Uptime
- Housekeeping

These basic principles relate directly to manufacturing processes, but they can also be tailored for the software engineering process. Each of the eight elements will be defined, and the applicability to software engineering processes will be discussed.

Continuous Flow

Continuous flow is the simple, smooth movement of a product through a process. With continuous flow, there are no places for the product to sit and wait for the next operation. It also minimizes the number of hand-offs between workers, since each hand-off requires time. The flow is also based on small lot increments, since the larger the lot size, the more time is required as products sit while the remainder of the lot is being produced.

The primary impact of continuous flow on the software process is to change the focus from large projects to smaller, incremental development projects. The implementation plan for a software project should be based on small, incremental pieces, with each piece taken through to completion. These incremental pieces can be individual features or requirements, or the fix of a single defect. The incremental development forces the design, implementation, test, and documentation associated with a single change to be completed before the developer starts on another feature. If a new version has a number of new features, then the features are prioritized and implemented on an incremental basis, and a completed product is available to ship at any time.

A continuous flow process also avoids the common problems associated with "feature creep". If the releases of a software product contain many features and are released infrequently, there is a tendency to load the new version with too many features to take advantage of a release opportunity. However, if incremental releases are planned, then multiple versions of the product can be released quickly, and there can be better response to the rapidly changing needs of the users.

Multi-functional Workers

The multi-functional worker concept is the enhancement of the workforce to meet changing market, customer, and production demands. This goal is met by expanding employees' skills and competencies and removing barriers to flexibility. Thus, workers are knowledgeable about other functions and can be used to support others in those areas. It also implies that there are standardized processes which everyone follows for a specific task, which permits the interchangeability of the workers.

Although software engineering tasks may be more complex and require a longer learning curve than some other industries, there are some important process changes that can be made to facilitate multi-functional workers. A significant training investment may be needed to have all software engineers qualified in various development languages and tools. In addition, increasing the knowledge of the domain is essential to increase the flexibility of the software engineers.

Composition of team members can also contribute to a multi-functional workforce. A multi-functional team has the appropriate resources from the software development, quality assurance and end-user areas to complete the task. They work concurrently and together to ensure that the product moves with continuous flow.

Pull-through System

The primary focus of the pull-through system is that a product (intermediate or final) is not produced unless there is a need for it to be produced. There should be a simple signal which is the driving mechanism to produce the product.

A pull-through system requires that the process be understood and predictable, with well-defined entry and exit criteria for each phase of the process. High quality plans are produced specifying the desired ordering of the tasks. It also requires that defect-free products are produced at every stage, so that the entire process will not have to wait for defects to be corrected. A characteristic of a pull-through system is that there are a limited number of products in the stream at any one time. The number of products is kept small and as one product is completed, another is added to the pipeline.

The pull-through system allows one to prioritize the features or products and then implement them incrementally. Only those features or products which are truly required are produced, and then only when needed. To ensure that there are clear transitions between stages and to reduce defects being passed to the next stage, the product is inspected before moving on to the next phase

Level Production

Level production is the establishment of smooth, uniform production flow to assure reliability and flexibility in meeting customer demand. The key feature of level production is to keep the production load as level as possible and to avoid the last minute crunches which tend to impair quality. Level load requires production planning and continuous flow.

Accurate high-level project planning is a key requirement of level production. For our software applications, the estimates of effort and time must be accurate and the status of project milestones must be carefully monitored. Scheduling of resources should allow for contingencies, with no more that 80% of a person's time being accounted for in the plan to allow time for the unexpected.

Level production can be facilitated when other JIT principles are applied. For example, with continuous flow, incremental development, multi-functional workers and reduced setup time, more flexibility is possible for load leveling between different projects.

Setup Time

Setup time is the elapsed time from production of the last good piece on the "old" setup to the first good piece on the "new" setup. To establish an efficient pull-through system, the setup time must be eliminated or dramatically reduced. Minimizing setup implies that change-overs required for different products or different stages of a product can be handled expeditiously. This means that the change-over process is well-defined, and if possible, as much of the change-over is handled off-line.

Setup time is applicable to several areas in software engineering. When moving to a new project, there is setup time for the software engineer to become familiar with the new development environment. The project work areas can be standardized, so that a person can easily transition to a new project and knows where to find all the important information. This implies a common development environment and configuration management process for each project. In addition, clear, complete, and readily available documentation of the software project will enable a new developer to join the project team and readily contribute.

In addition, all process changes and tool changes should be carefully planned and controlled to minimize the impact on the developer. Guidelines and templates should be prepared, and appropriate training provided.

Product Quality

Product quality is essential to JIT. The goal is to achieve zero defect product quality performance through continuous quality improvement. Systematic defect searches are performed and statistical process control is used to keep the quality of the product as high as possible. If a problem is discovered, the source of the problem is fixed immediately.

This particular element of JIT is embodied in the CMM level 5. Defects are found at the source and the root cause is determined and fixed.

Machine Uptime

To achieve maximum cycle time reduction and continuous flow, a goal of 100% on demand equipment utilization should be attained. This requires systematic inspection, detection and prevention of equipment failure in production.

Within software engineering, this element of JIT includes the entire development environment, including process, tools, and system hardware. The CMM addresses only the process side, yet the hardware is also very critical for software engineers. The computer network should be configured for reliability with hardware acquired to ensure uptime (e.g., uninterrupted power supplies, reliable disk technology such as RAIDs). Procedures have been implemented to ensure disaster recovery.

Housekeeping

Housekeeping involves the attainment of a clean and orderly workplace, to improve work flow, improve morale, improve safety, and enhance product and process quality.

For a software engineer, the most important housekeeping item is the configuration management system. All the projects should use a standard structure for keeping all the files associated with the project. Coding standards, naming conventions, standard document formats, re-use standards, and other good software engineering practices can also contribute towards an orderly development environment.

Seven Deadly Wastes

To help in determining focus areas for process improvements, several common areas have been identified as being the most likely areas for improvement. These seven areas where non-value added time and process wastes are typically uncovered in implementing TBM are:

- Over Producing
- Time on Hand Waiting
- Transporting

- The Process Itself
- Unnecessary Stock on Hand
- Unnecessary Motion
- Production of Defective Goods

The basic principles of the Seven Deadly Wastes were derived from manufacturing processes, but they can also be interpreted for the software engineering process. Each of the seven wastes are described below, along with how they can be applied to the software engineering process.

Over Producing

Producing more than what is needed is a waste of resources. The extra product will have to be stored or thrown out. Many times, software products are developed with the familiar "bells and whistles", which may not be required or even desired by the customer. It is also important to have clear prioritization of the requirements to ensure that the most important user features are developed. Or how many times have we seen a software engineer not reuse code because he could do it better.

Time on Hand Waiting

Lost time can result from simply waiting for the next step in the process. The time delays between phases can slow down a software project. Lack of appropriate resources (such as people, hardware, or tools) through poor planning can result in waiting and delays. Document review, approval processes, and waiting for tester availability may also create delays.

Transporting

Transporting is defined as the time spent in moving the product from one place to another between stages. This can occur due to the separations or boundaries which impede the smooth flow of information or products. An excellent example is the use of uncoupled tools at various phases which do not communicate with each other.

The Process Itself

Studies have shown that typically 95% of the elapsed time on a project is "wasted" or does not add value from a customer perspective. For example, defect detection, defect removal and rework imply wasted effort. Thus, any process contains significant wasted time that can be removed via aggressive cycle-time reduction efforts. The software process should be continuously reviewed to determine inefficiencies in the process. The key to success in this area is to separate process improvement from production and to dedicate a few individuals to lead the process improvement changes.

Unnecessary Stock on Hand

Creating stock which is stored before the next phase or being shipped to the customer is another time waster. An example from a software engineering perspective is the feature laden code which can't make it through testing. Incremental software development helps to reduce the stock on hand, especially when used in conjunction with rapid release cycles. Another temptation that has to be resisted is the "Wouldn't it be neat if ..." syndrome where developers get side-tracked onto intellectually stimulating, but unnecessary functions.

Unnecessary Motion

Unnecessary motion means the unneeded movement of the product during the production or delivery. For electronic products, this can occur if the process requires the movement between different directories or the renaming of files. Also, just having excessive reviews or an unnecessary person in the loop can be a bottleneck. Lost time due to unnecessary motion can also be subtle items such as electronic barriers which make it more difficult to transfer information.

Production of Defective Goods

Anytime a defective good is produced, it has an impact on the cycle-time, since it has to be fixed or produced again. The timely detection of defects and the root-cause analysis of the defects are important to minimize this waste.

Time Based Management Implementation Experience

A systematic approach to TBM implementation within the Westinghouse CNFD organization has resulted in significant improvements to the software process. Cycle time has reduced, new features are available to the users at an earlier date, and quality has improved.

The steps for implementing TBM include:

- Training and sponsorship
- Process mapping
- Development of a TBM action plan
- Implementation of key actions
- Results

Along each step of the process, management commitment to TBM was evident and clearly communicated to all personnel.

Training and Sponsorship

A comprehensive training program was needed to begin TBM implementation. First, awareness training sessions were held with all employees. This introduced the TBM concepts, reinforced the TQM foundation of our organization, and presented the business case for changing operational strategies. A key here was to focus on the benefits of TBM and emphasize that we were not talking about making people work faster, rushing or sacrificing quality for schedule. Sponsorship training was conducted by working with the departmental management staff to educate them on the academic basis for the TBM concepts and to discuss the various business case studies. They needed to be strong sponsors for successful change. Finally, all employees received training on the Eight Elements of Just-In-Time and the Seven Deadly Wastes. This led to actual process mapping sessions and process re-design implementation.

Process Mapping

In implementing TBM, the first activity after training was to map the current process. The process map was based on a simple time line of events. The event was described in a verb - predicate phrase (i.e., Issue requirements). For each event, the time that it actually took for a typical product as well as the elapsed time were defined. Additional comments on the process (clarifications or questions) were also added.

From the time line, a work-time profile was created by plotting the integral of the work effort versus the elapsed time. Also determined was the amount of wait time compared to the total elapsed time. These items tended to identify the dead periods in the project and help to clarify the potential for cycle-time reduction.

The team was then asked to create a new process map which showed a substantial reduction in elapsed cycle-time. A 50% reduction in cycle time is typical. This was accomplished by studying the key elements of Just-In-Time and the seven deadly wastes and determining what events could be eliminated or reduced to cut the cycle-time. The new process map was developed and process changes identified. These changes were validated with a peer group not involved in the original re-design.

Development of a TBM Action Plan

Based on the process mapping, an action plan was documented for TBM items. Each action item was categorized by the effort required to implement, as well as the potential pay-back in CTR which could result. Those items which could be implemented quickly and easily, and would result in a significant reduction in cycle time, were give the highest priority. This plan was published to all employees. A monthly status is also published and accomplishment metrics are maintained.

Implementation of Key Actions

The organization was rearranged to separate process from production and form a dedicated process group. This group was committed to implementing high priority items from the TBM action plan. Also, a TBM application team with members from all impacted software groups was formed. This team helped focus on completing the TBM action plan. Process changes were documented, inspected, and incorporated into a software methodology manual. Training was provided on all process changes, with monthly seminars being held to review current process changes.

Although a software methodology manual already existed, it had been largely unused on a day-to-day basis by the software engineers. Increased visibility was given to this document by providing more frequent training, and using management reinforcement. In addition, the document was made available on-line to provide easy access to the ongoing process changes.

The CMM and TBM

The Capability Maturity Model is consistent with the principles of TBM, since both the CMM and TBM are both based upon Total Quality Management. But TBM does provide a number of new insights into good software practices that are not obvious from the CMM. A comparison of the CMM Key Process Areas (KPAs) with the implementation suggestions for TBM in software engineering is presented in Table 1. This shows the correlation and extensions to the CMM for the cycle-time perspective.

Table 1: Correlation between CMM and TBM

CMM KPA[8,9]	TBM Correlation
Software Project Planning	Production leveling requires accurate project estimates and plans. Large projects are broken up into small, easily managed parts. Multi-functional workers provide more flexibility for resource management.
Software Project Tracking (V1.1) Software Project Control (V2.0)	The project is tracked and resources are adjusted within the project to keep the project flowing smoothly.
Software Subcontract Management (V1.1) Software Acquisition Management (V2.0)	Cross-functional, integrated teams require the sub-contractors to work closely on the project with frequent interactive communications.
Software Quality Assurance	Software products have quality checks throughout the process to quickly eliminate defects which minimize their impact.
Software Configuration Management	Housekeeping requires configuration management of all software products. The standard desk-top defines a common directory structure within the configuration management system used by all projects for all documentation and source.
Requirements Management	Incremental development promotes clear and concise requirements for the specific incremental effort. The appropriate dedicated resources for the task are assigned to enable rapid completion of the project. The ability to quickly release a new version enables the project manager to "say no" to late requirements.
Organization Process Focus	Continuous process improvement is inherent in Cycle - Time Reduction. Processes are mapped and then re-designed to reduce cycle time.
Organization Process Definition	Process mapping has been performed and analyzed for wasted cycle time. New processes have been formally defined and are periodically reviewed for further enhancements.
Peer Reviews	Software products are checked for quality throughout the process to quickly eliminate defects to minimize the their impact. Small, cross-functional teams work closely together to produce the product.
Training Program (V1.1) Organization Training Program (V2.0)	Multi-functional workers have been trained to perform their functions and associated functions.
Intergroup Communication (V1.1) Project Interface Coordination (V2.0)	Cross-functional, integrated teams are utilized to avoid communication bottlenecks.

Software Project Engineering	Standards are defined for the location and structure of intermediate products to ensure consistent application and worker flexibility.
Integrated Software Management	With the focus on small projects, the organization as a whole focuses on an integrated, prioritized project plan to enable the level flow of work through the organization. Risks associated with cycle time must be managed aggressively.
Quantitative Process Management (V1.1) Statistical Process Management (V2.0)	Cycle-time metrics are added and tracked. Focus is on reducing the cycle-time and is visible to all layers of management. The cycle-time metric is a global metric that can easily be implemented in every organizational unit and understood by all involved.
Software Quality Management (V1.1) Organization Process Performance (V2.0)	The organization maps the software process looking for cycle-time reductions. It then tracks cycle-time and quality measures throughout the life-cycle to determine how to improve the process.
Organization Software Asset Commonality (V2.0)	Re-use is strongly supported to reduce cycle-time by eliminating repetitive development. Standards are defined for the location and structure of intermediate products to ensure consistent application and worker flexibility.
Defect Prevention	Cycle-time reduction looks for repetitive defects and seeks to eliminate the cause.
Technology Innovation (V1.1) Organization Process & Technology Innovation (V2.0)	Cycle-time is also used as a metric to determine the effectiveness of technology innovation. The software process is mapped and analyzed to determine areas for cycle-time reduction.
Process Change Management (V1.1) Organization Improvement Deployment (V2.0)	The implementation of process changes or new tools are carefully orchestrated to minimize the startup time. Guidelines for the usage of new tools are provided to enable the rapid assimilation of the new capabilities.
Support Systems Management (not in CMM)	The hardware network is designed and managed to provide maximum up-time for development. Software tools are similarly managed to provide a dependable work environment.

Results

Since TBM is based upon TQM, we first focused on rectifying the organization's deficiencies in the CMM KPAs. We also identified a number of simple, yet powerful changes in the process which are directly related to cycle-time reduction. We laid out an implementation plan and are still following that plan. Given the overall organizational focus, there was funding made available for process changes and follow-up which has been critical for our success. The movement has been swift and the institutionalization of the new process is happening at a rapid rate.

Significant changes have occurred in our software process due to our focus on reducing cycle-time and implementing TBM. About 25% of our software releases now are completed in 10 days or less while this was not even considered as feasible under the old system. The overall average cycle-time has been cut over 75% from the base process maps. Our software first-time yield has climbed to an average of 84% YTD compared to under 70% pre-TBM. Standards in languages, databases, GUIs, directory structures and re-use have all been developed and implemented. Detailed metrics have been established and are driving new process changes. Over half of the software engineers are on a path to be multi-functional in software (either language or specific code application). Software inspections are now being held at each phase of the software life-cycle for each project. These have contributed to our increased quality (yield). Also a cadre of outside software contractors are no longer used as the internal workforce has become more productive in the state-of-the-art tools and languages. This has led to significant bottom-line budget savings. All of this was accomplished in less than 20 months starting with the initial process mapping and training. By focusing on cycle-time reduction, we have seen our software product quality improve, our productivity increase, our costs become lower and our software maturity level rise. It is not clear we would have achieved similar results by focusing on any other areas in this way. TBM has taught us that time is a powerful lever in your organizational toolbox. We plan to keep moving on our TBM journey.

Conclusions

The principles of time-based management are directly applicable to software engineering. They provide a different way of looking at the process and as such can yield significant improvements. Just as implementing TQM process changes showed significant cycle-time reductions, implementing TBM process changes can also lead to significant quality improvements. But the TBM perspective has more of a business perspective and can show higher returns than just the TQM perspective. These process changes therefore have a greater chance of getting management support. For those already at a high maturity level, it provides incentives for even more process changes. For those at the lower maturity levels, it provides a compelling business reason to improve the process maturity level.

References

1. Cusumano, M. & Selby, R., *Microsoft Secrets - How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, The Free Press, 1995, ISBN 0-02-874048-3.
2. Japan Management Association, ed., *Kanban - Just-in-Time at Toyota*, Productivity Press, 1986, ISBN 0-915299-48-8.
3. Krasner, H., *Software Cycle-Time Reduction (SCTR): Boldly Going Beyond Level 3*, 8th SEPG Conference, Atlantic City, NJ, May 1996.
4. Maguire, S., *Debugging the Development Process*, Microsoft Press, 1994, ISBN 1-55615-650-2.
5. McConnell, S., *Rapid Development - Taming Wild Software Schedules*, Microsoft Press, 1996, ISBN 1-55615-900-5.
6. McFeeley, B., *IDEAL: A User's Guide for Software Process Improvement*, CMU/SEI-96-HB-001, Software Engineering Institute, 1996.
7. Meyer, C., *Fast Cycle Time - How to Align Purpose, Strategy, and Structure for Speed*, The Free Press, 1993, ISBN 0-02-921181-6.
8. Software Engineering Institute, *The Capability Maturity Model Guidelines for Improving the Software Process*, Addison-Wesley, 1995, ISBN 0-201-54664-7.
9. Software Engineering Institute, *SW-CMM Version 2.0*, Draft C, October 22, 1997
10. Stalk, G. & Hout, T., *Competing Against Time - How Time-Based-Competition Is Reshaping Global Markets*, The Free Press, 1990, ISBN 0-02-915291-7.

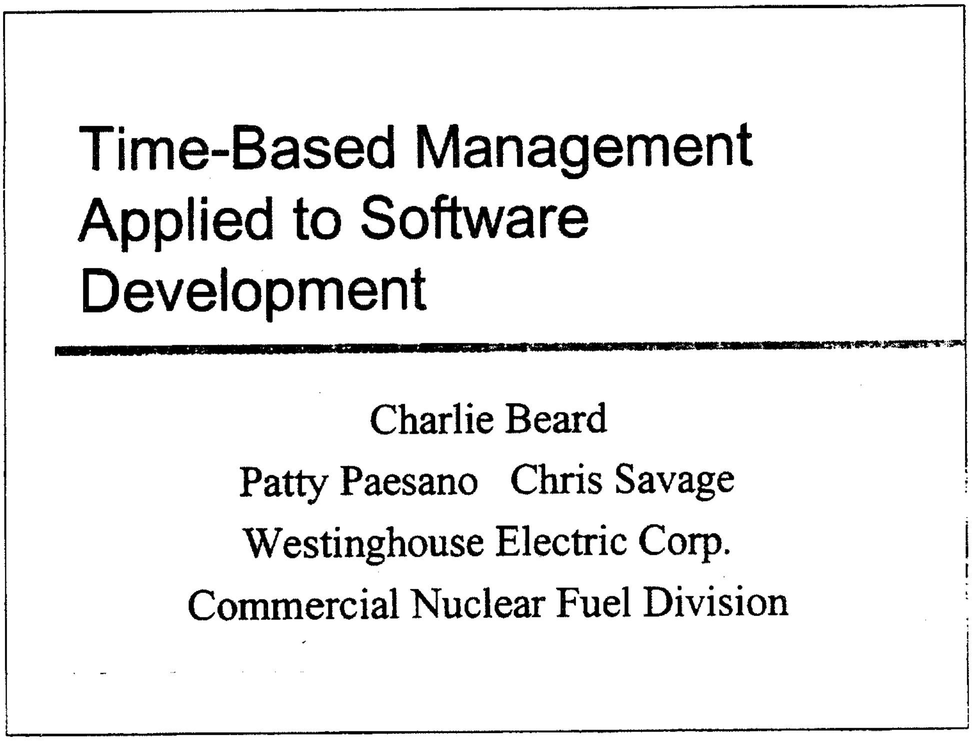
Contact Information

Charlie Beard
beardcl@westinghouse.com
(412) 374-2164

Patty Paesano
paesanpc@westinghouse.com
(412) 374-2649

Chris Savage
savagecr@westinghouse.com
(412) 374-2290

Commercial Nuclear Fuel Division
Westinghouse Electric Corporation
P. O. Box 355
Pittsburgh, PA 15230-0355



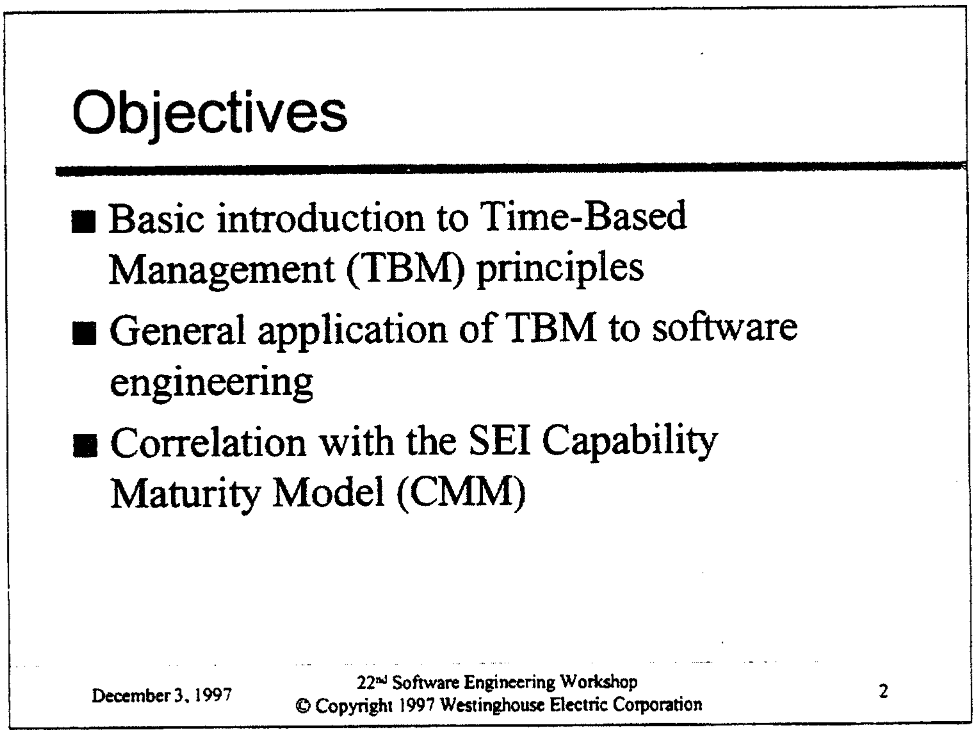
Time-Based Management Applied to Software Development

Charlie Beard

Patty Paesano   Chris Savage

Westinghouse Electric Corp.

Commercial Nuclear Fuel Division



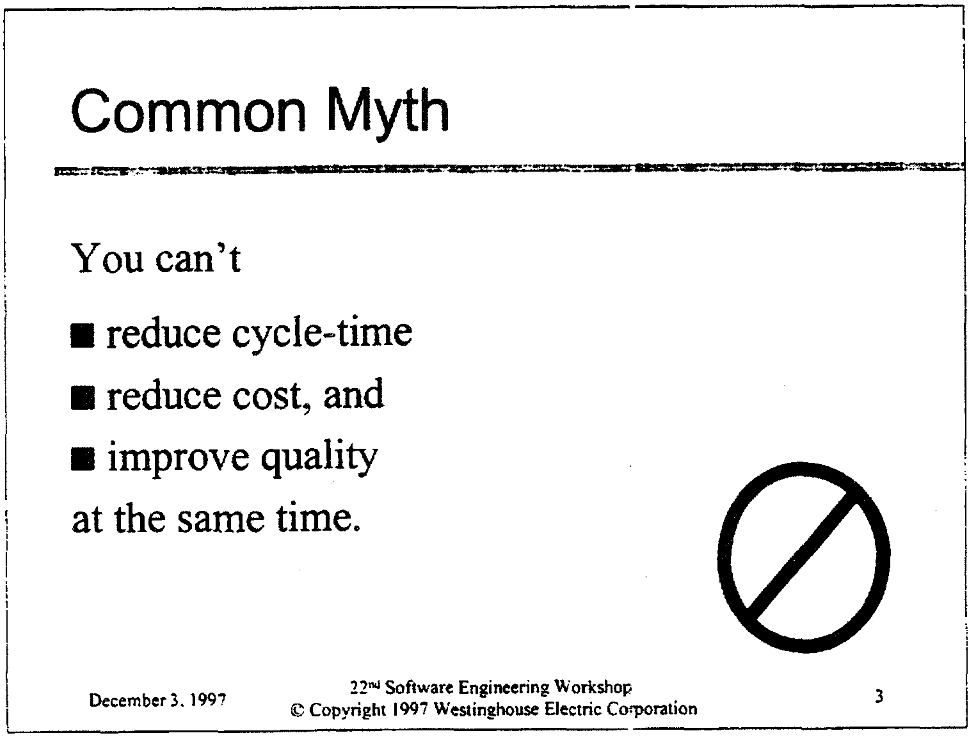
Objectives

- Basic introduction to Time-Based Management (TBM) principles
- General application of TBM to software engineering
- Correlation with the SEI Capability Maturity Model (CMM)

December 3, 1997

22nd Software Engineering Workshop

2



Common Myth

You can't

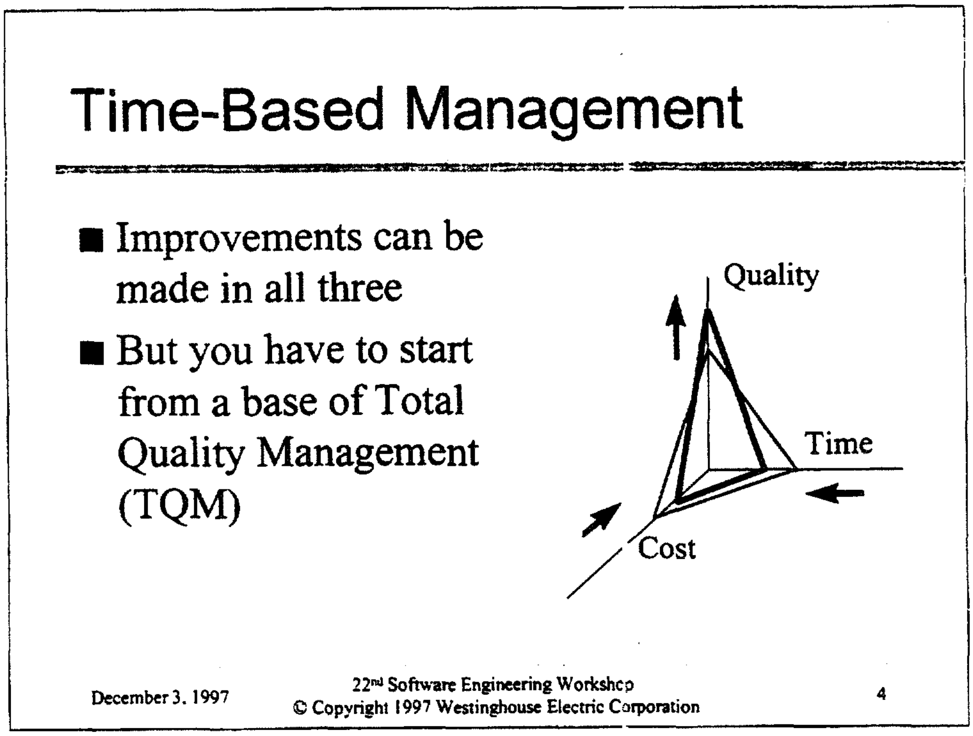
- reduce cycle-time
- reduce cost, and
- improve quality

at the same time.

December 3, 1997

22nd Software Engineering Workshop

3



Time-Based Management

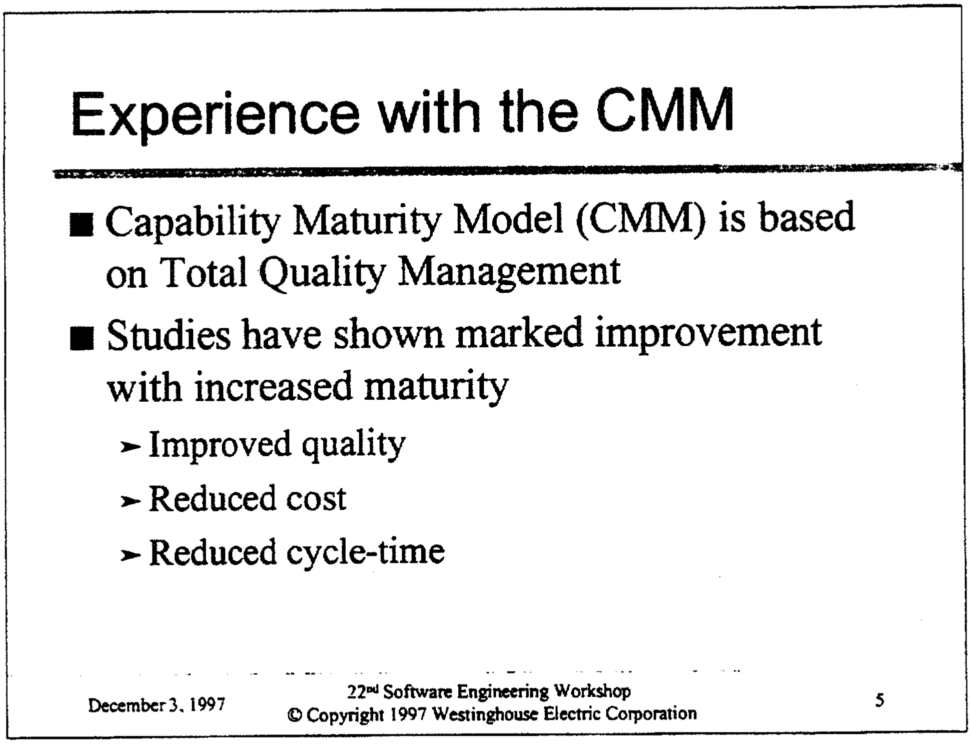
- Improvements can be made in all three
- But you have to start from a base of Total Quality Management (TQM)

Quality
Time
Cost

December 3, 1997

22nd Software Engineering Workshop

4



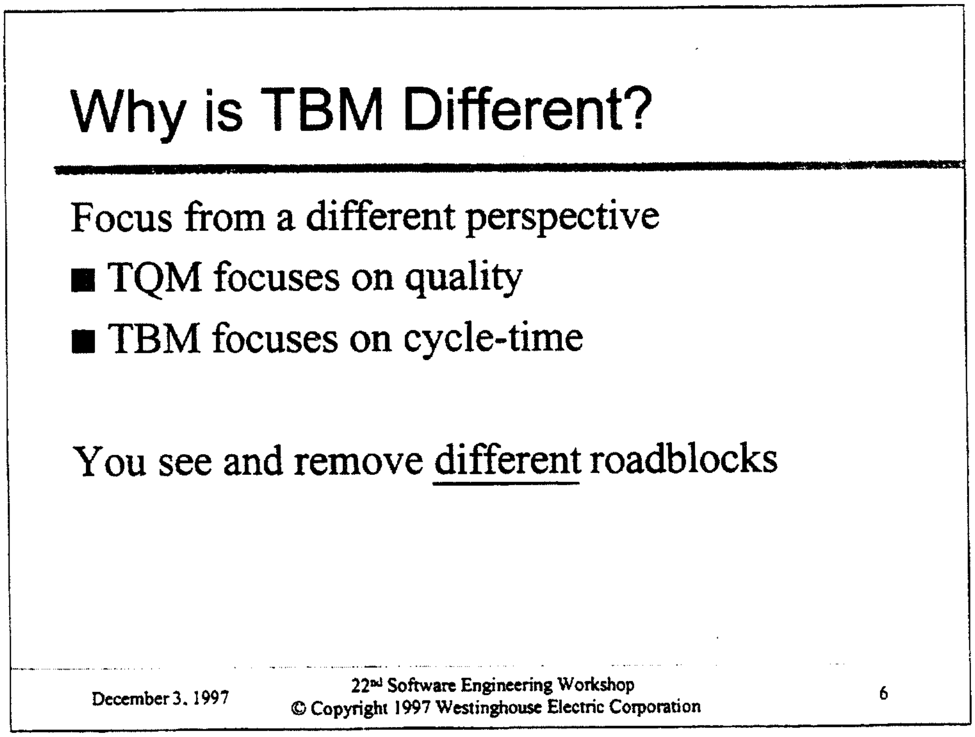
Experience with the CMM

- Capability Maturity Model (CMM) is based on Total Quality Management
- Studies have shown marked improvement with increased maturity
  - ➢ Improved quality
  - ➢ Reduced cost
  - ➢ Reduced cycle-time

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

5



Why is TBM Different?

Focus from a different perspective

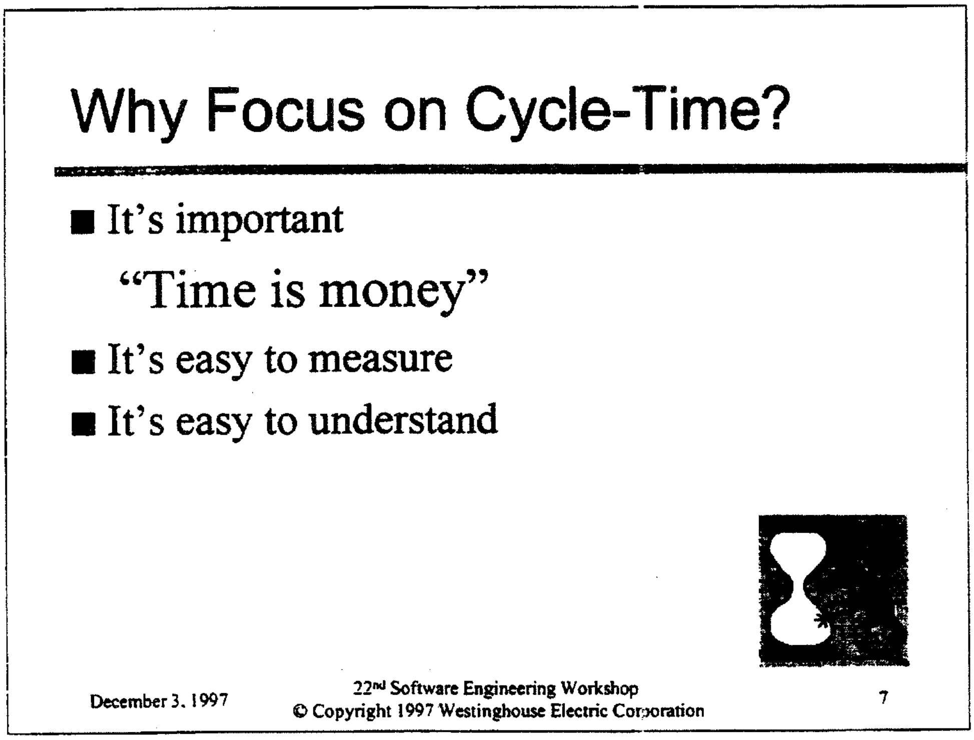
- TQM focuses on quality
- TBM focuses on cycle-time

You see and remove different roadblocks

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

6



Why Focus on Cycle-Time?

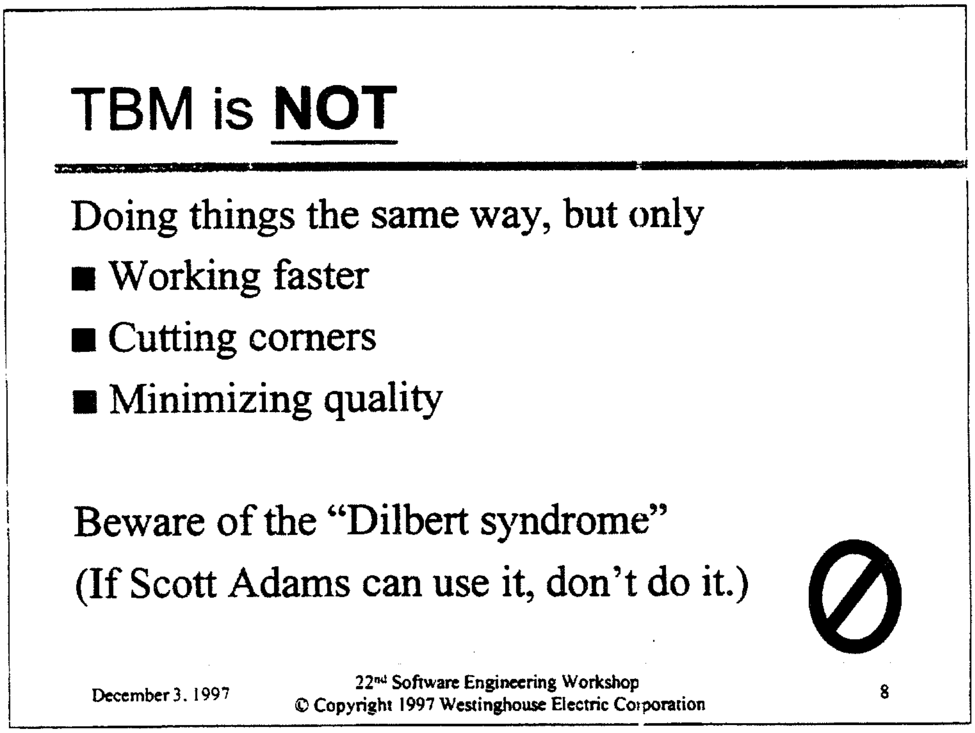
- It's important

  "Time is money"
- It's easy to measure
- It's easy to understand

December 3, 1997

22nd Software Engineering Workshop

7



TBM is NOT

Doing things the same way, but only

- Working faster
- Cutting corners
- Minimizing quality

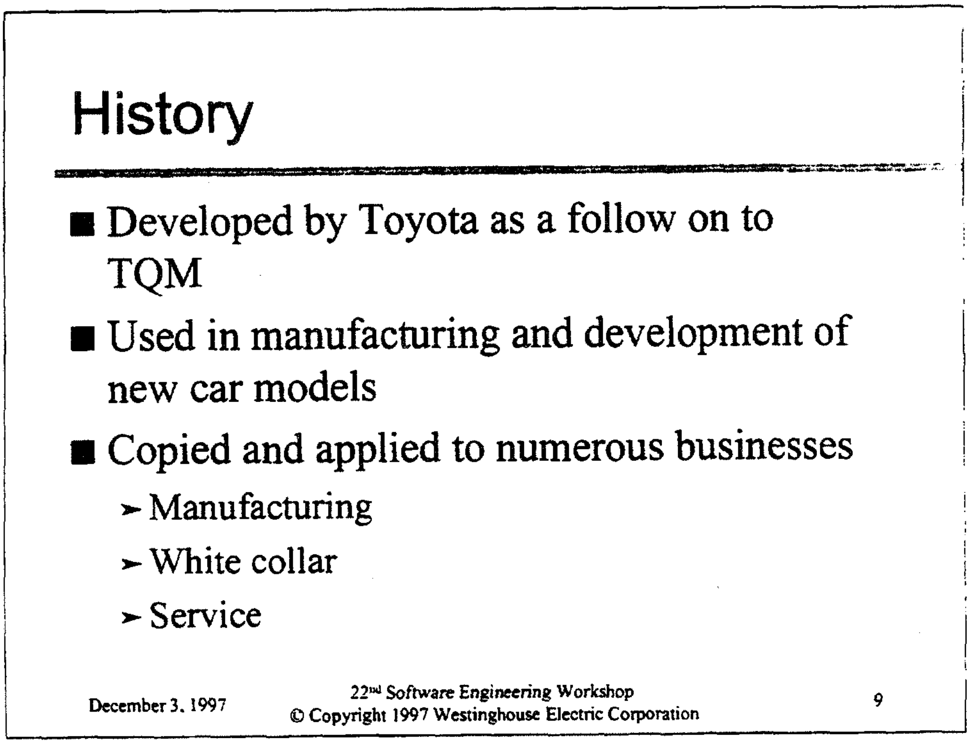
Beware of the "Dilbert syndrome"

(If Scott Adams can use it, don't do it.)

December 3, 1997

22nd Software Engineering Workshop

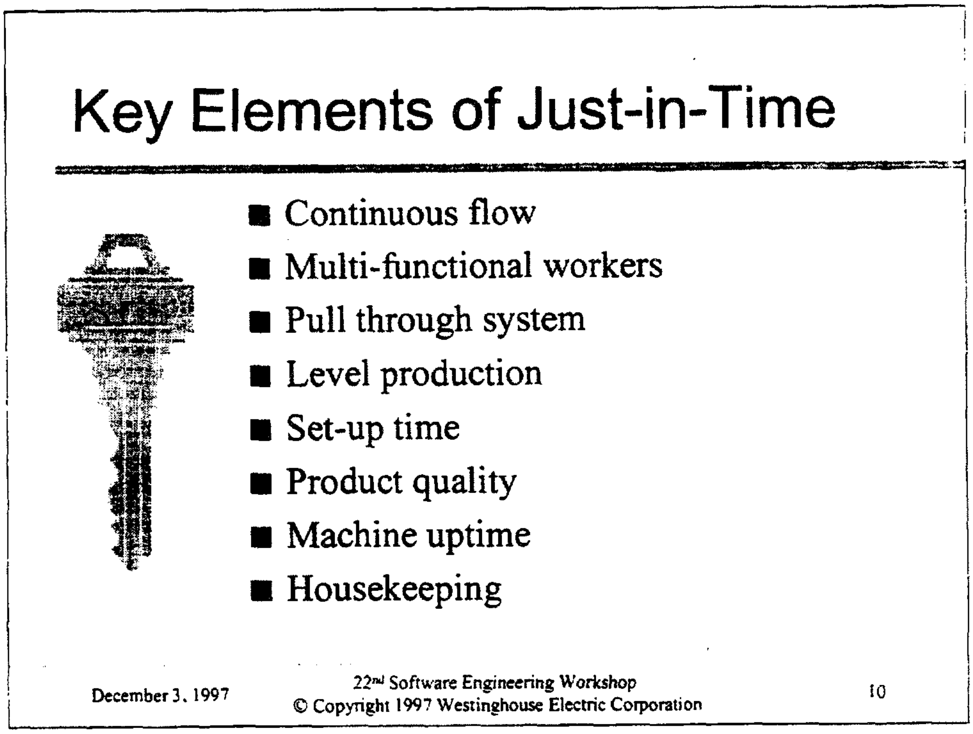
8



History

- Developed by Toyota as a follow on to TQM
- Used in manufacturing and development of new car models
- Copied and applied to numerous businesses
  - ➢ Manufacturing
  - ➢ White collar
  - ➢ Service

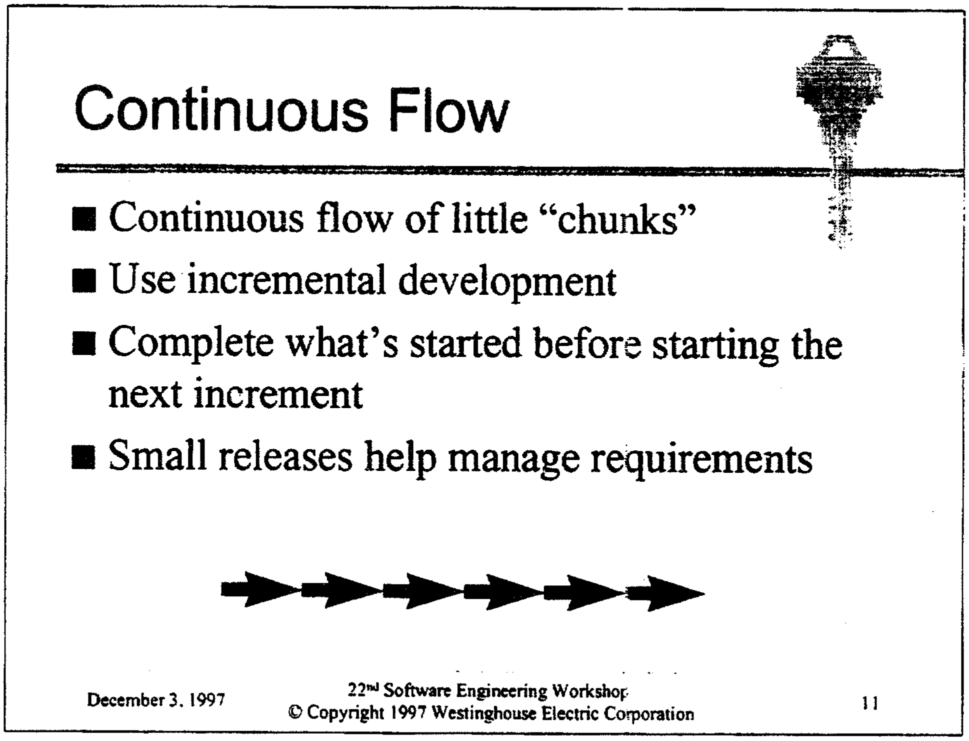
December 3, 1997 22nd Software Engineering Workshop © Copyright 1997 Westinghouse Electric Corporation 9



Key Elements of Just-in-Time

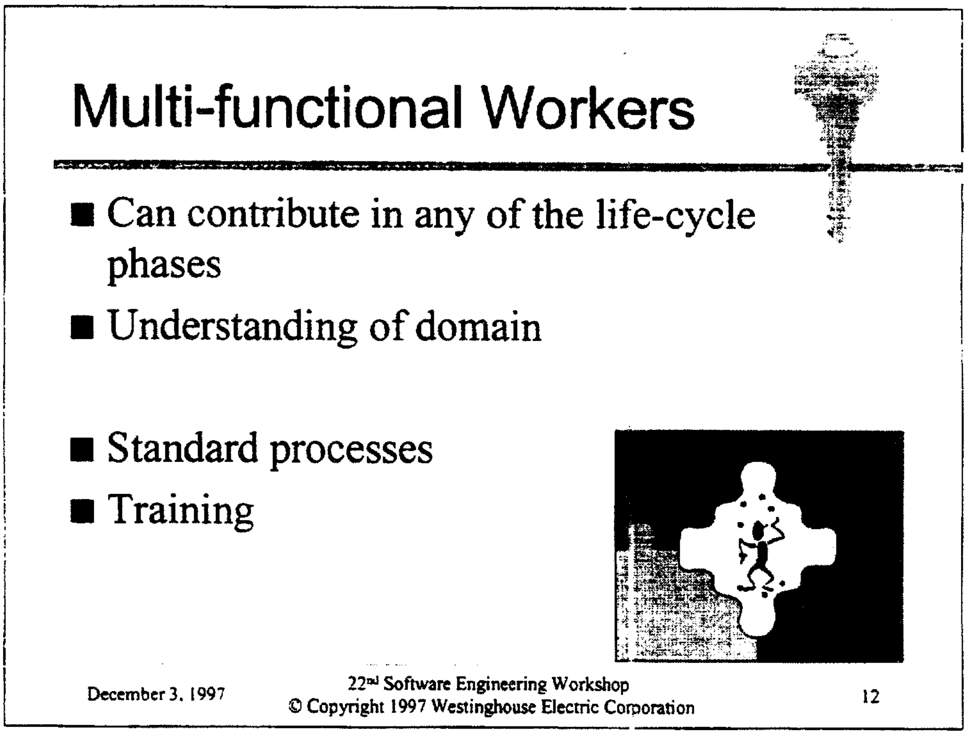
- Continuous flow
- Multi-functional workers
- Pull through system
- Level production
- Set-up time
- Product quality
- Machine uptime
- Housekeeping

December 3, 1997 22nd Software Engineering Workshop © Copyright 1997 Westinghouse Electric Corporation 10



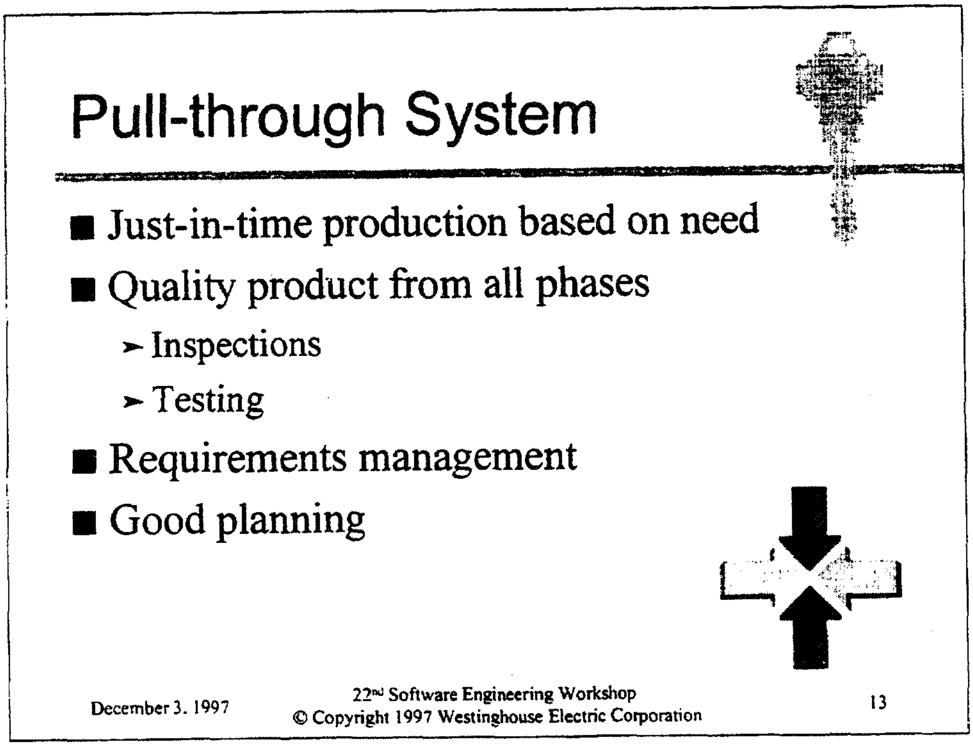
Continuous Flow

- Continuous flow of little "chunks"
- Use incremental development
- Complete what's started before starting the next increment
- Small releases help manage requirements



Multi-functional Workers

- Can contribute in any of the life-cycle phases
- Understanding of domain
- Standard processes
- Training

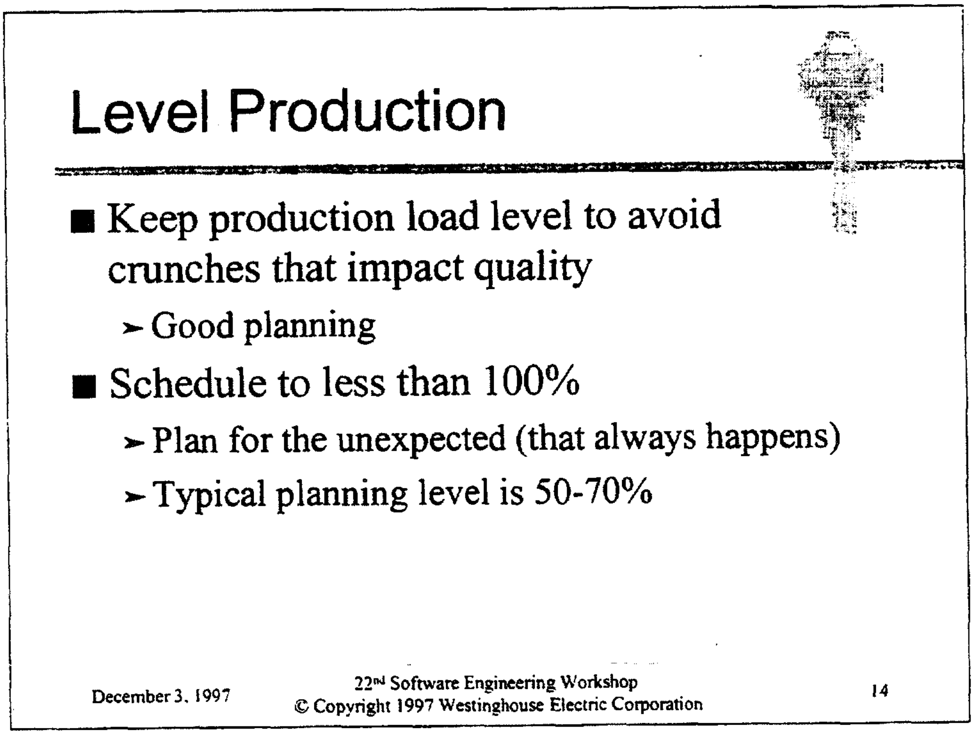


Pull-through System

- Just-in-time production based on need
- Quality product from all phases
  - Inspections
  - Testing
- Requirements management
- Good planning

December 3, 1997 22nd Software Engineering Workshop

13

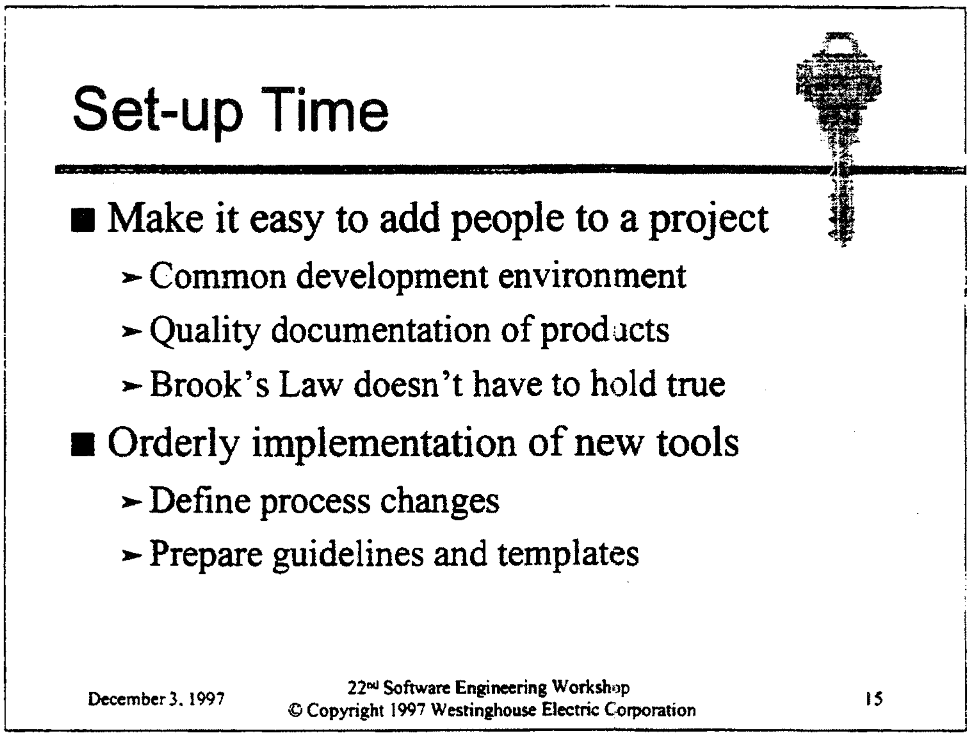


Level Production

- Keep production load level to avoid crunches that impact quality
  - Good planning
- Schedule to less than 100%
  - Plan for the unexpected (that always happens)
  - Typical planning level is 50-70%

December 3, 1997 22nd Software Engineering Workshop

14



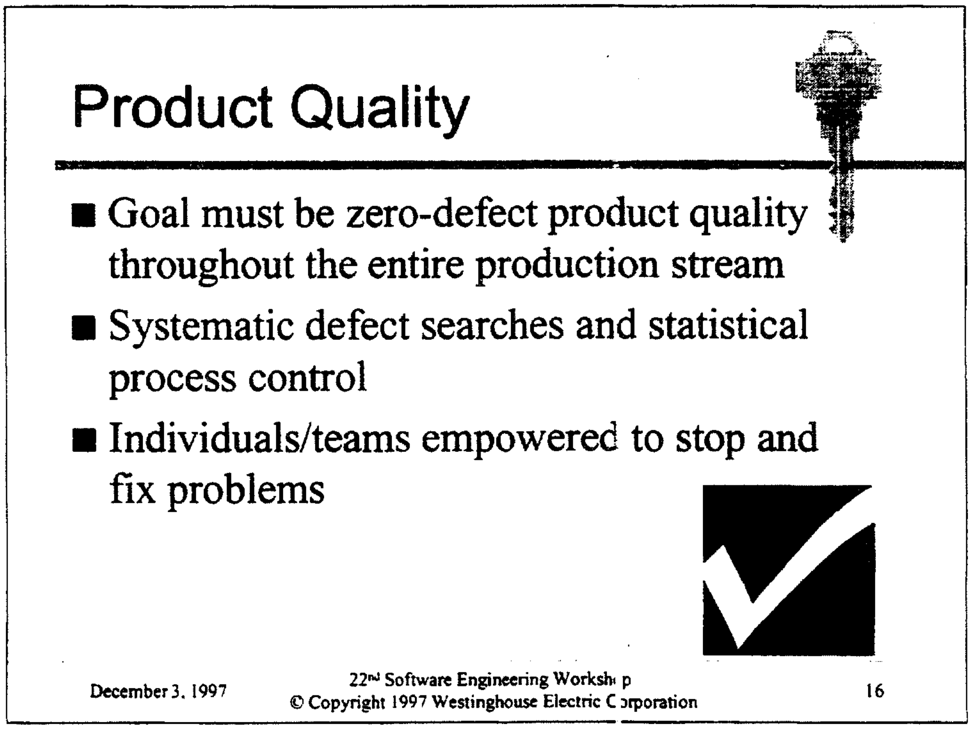
Set-up Time

- Make it easy to add people to a project
  - ➢ Common development environment
  - ➢ Quality documentation of products
  - ➢ Brook's Law doesn't have to hold true
- Orderly implementation of new tools
  - ➢ Define process changes
  - ➢ Prepare guidelines and templates

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

15



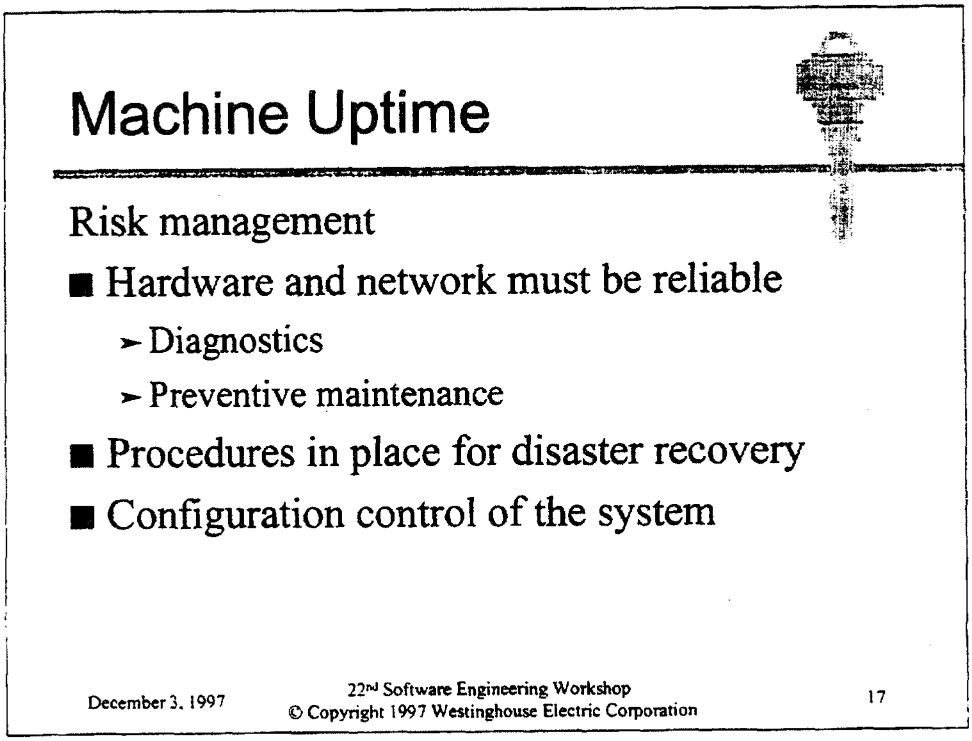
Product Quality

- Goal must be zero-defect product quality throughout the entire production stream
- Systematic defect searches and statistical process control
- Individuals/teams empowered to stop and fix problems

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

16



Machine Uptime

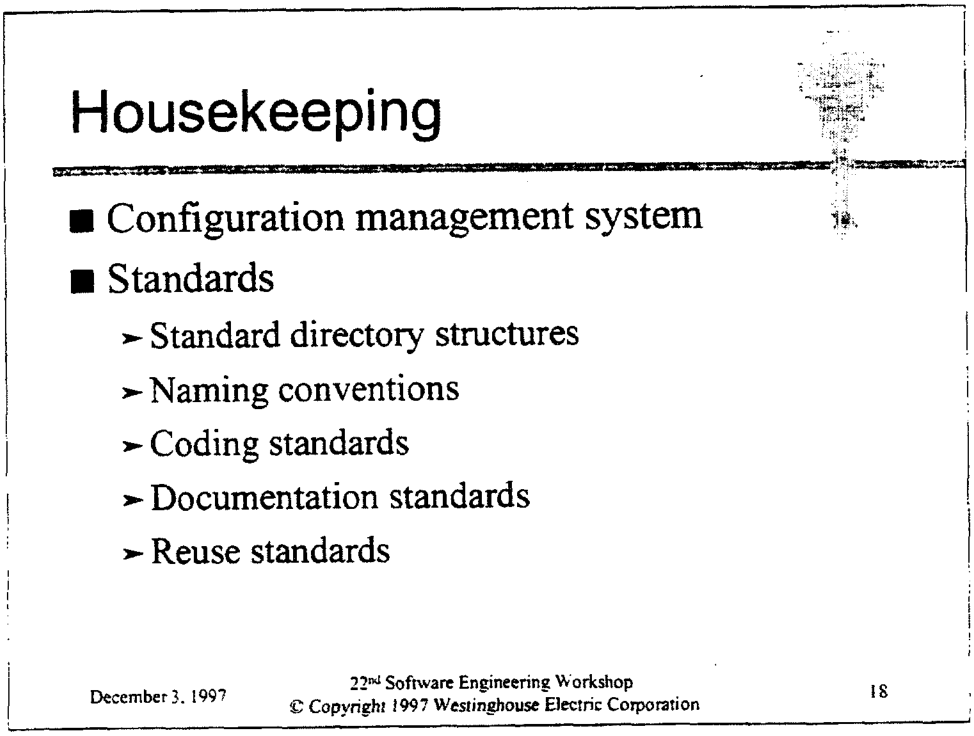
Risk management

- Hardware and network must be reliable
  - Diagnostics
  - Preventive maintenance
- Procedures in place for disaster recovery
- Configuration control of the system

December 3, 1997

22nd Software Engineering Workshop

17



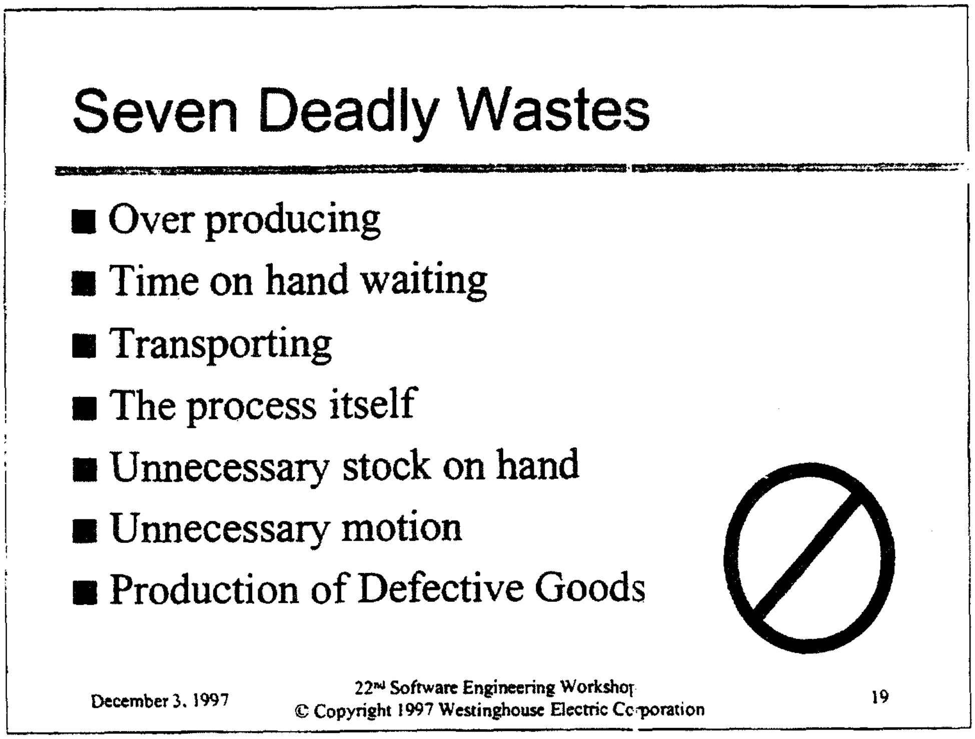
Housekeeping

- Configuration management system
- Standards
  - Standard directory structures
  - Naming conventions
  - Coding standards
  - Documentation standards
  - Reuse standards

December 3, 1997

22nd Software Engineering Workshop

18



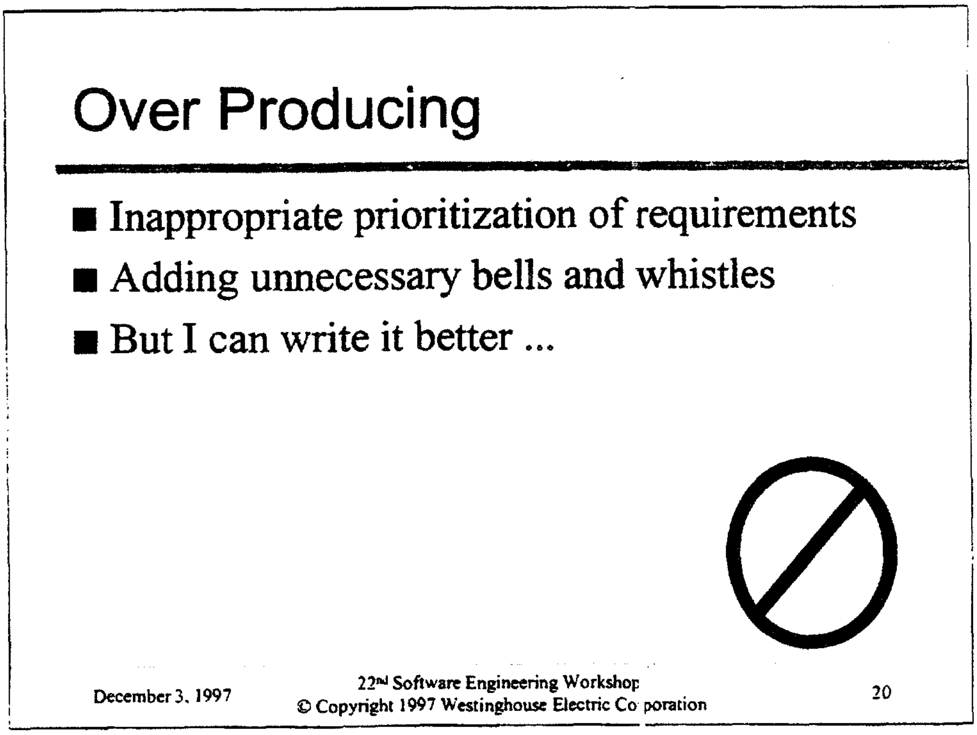
Seven Deadly Wastes

- Over producing
- Time on hand waiting
- Transporting
- The process itself
- Unnecessary stock on hand
- Unnecessary motion
- Production of Defective Goods

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

19



Over Producing

- Inappropriate prioritization of requirements
- Adding unnecessary bells and whistles
- But I can write it better ...

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

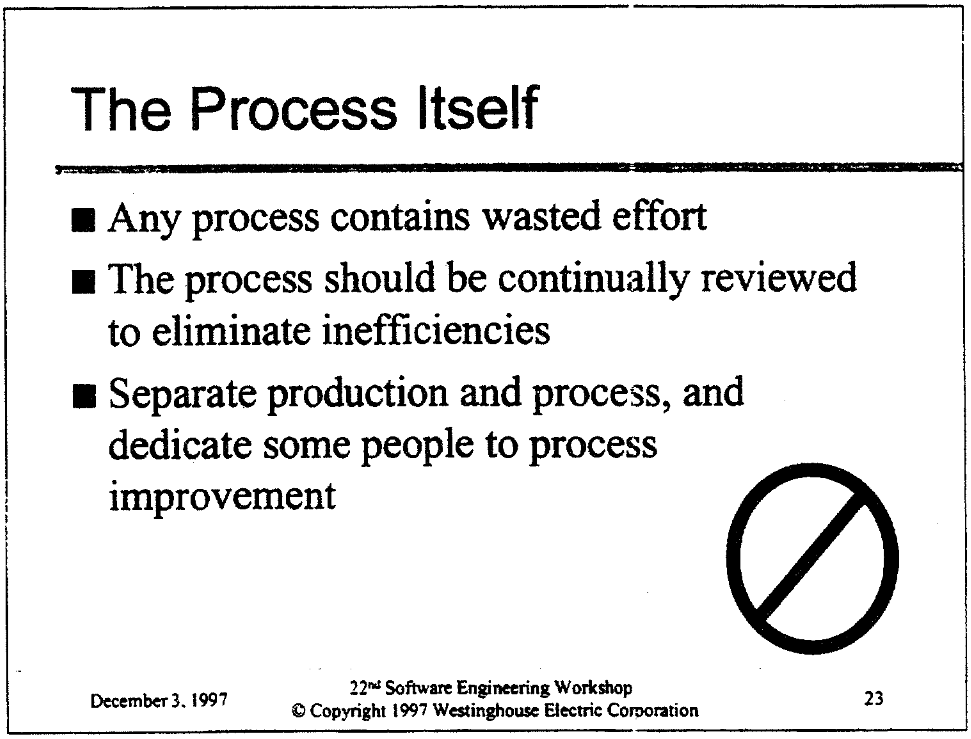
20

Time on Hand Waiting

- Inappropriate scheduling of tasks
  - inspections
  - testing
  - documentation
  - the ambiguous front-end
- Task sitting in a queue

Transporting

- Time spent in moving products between stages
- Electronic barriers



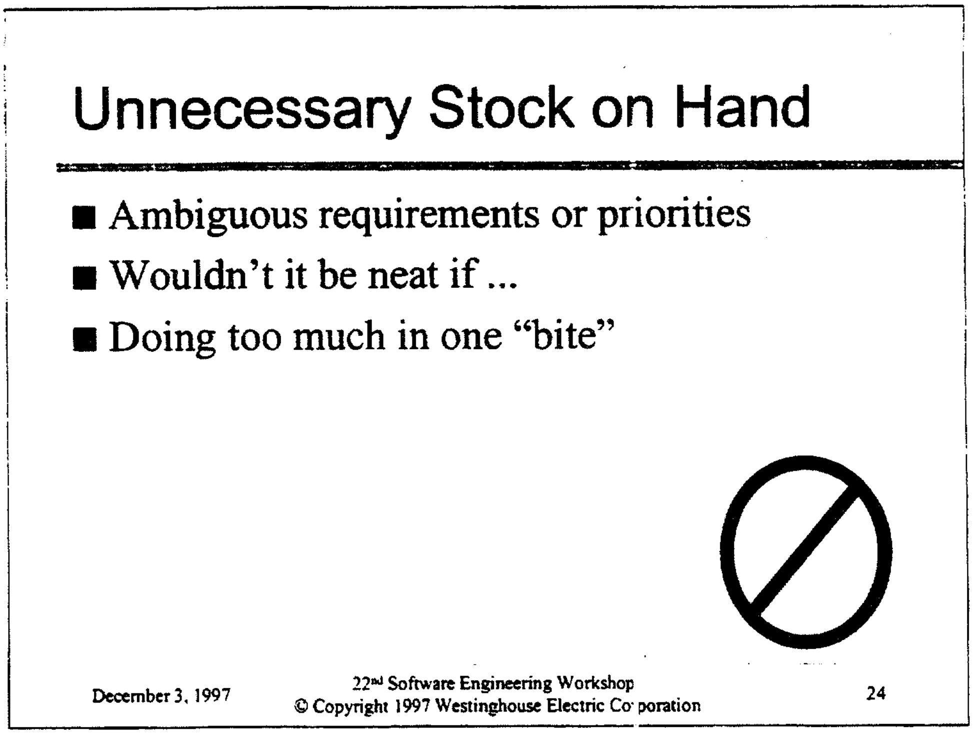
The Process Itself

- Any process contains wasted effort
- The process should be continually reviewed to eliminate inefficiencies
- Separate production and process, and dedicate some people to process improvement

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

23



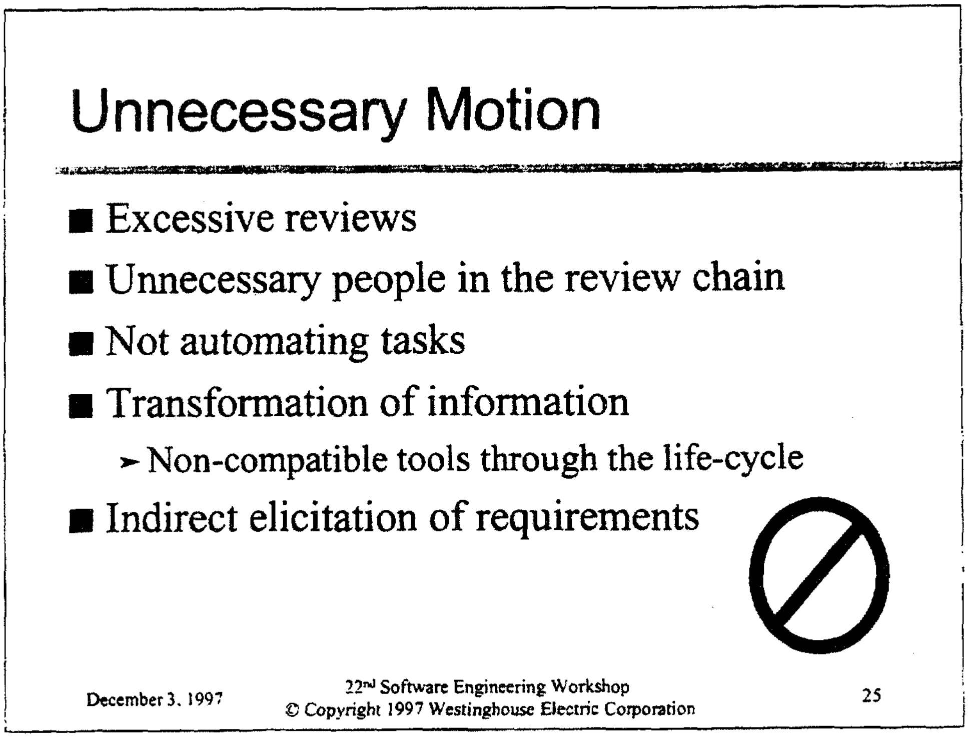
Unnecessary Stock on Hand

- Ambiguous requirements or priorities
- Wouldn't it be neat if ...
- Doing too much in one "bite"

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Co poration

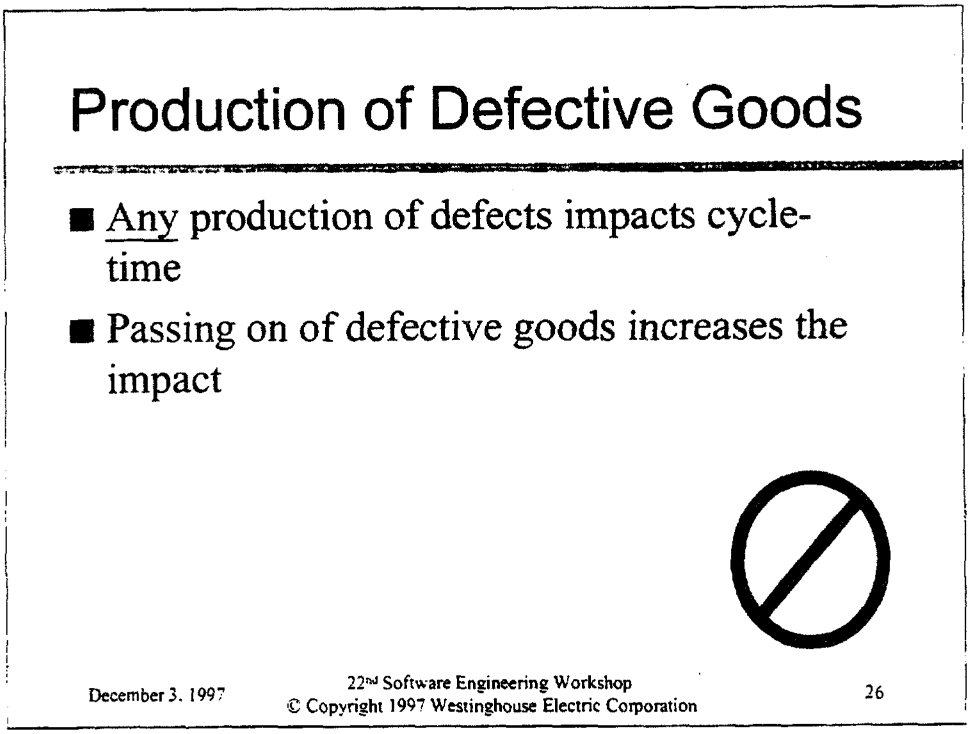
24



Unnecessary Motion

- Excessive reviews
- Unnecessary people in the review chain
- Not automating tasks
- Transformation of information
  - ➢ Non-compatible tools through the life-cycle
- Indirect elicitation of requirements

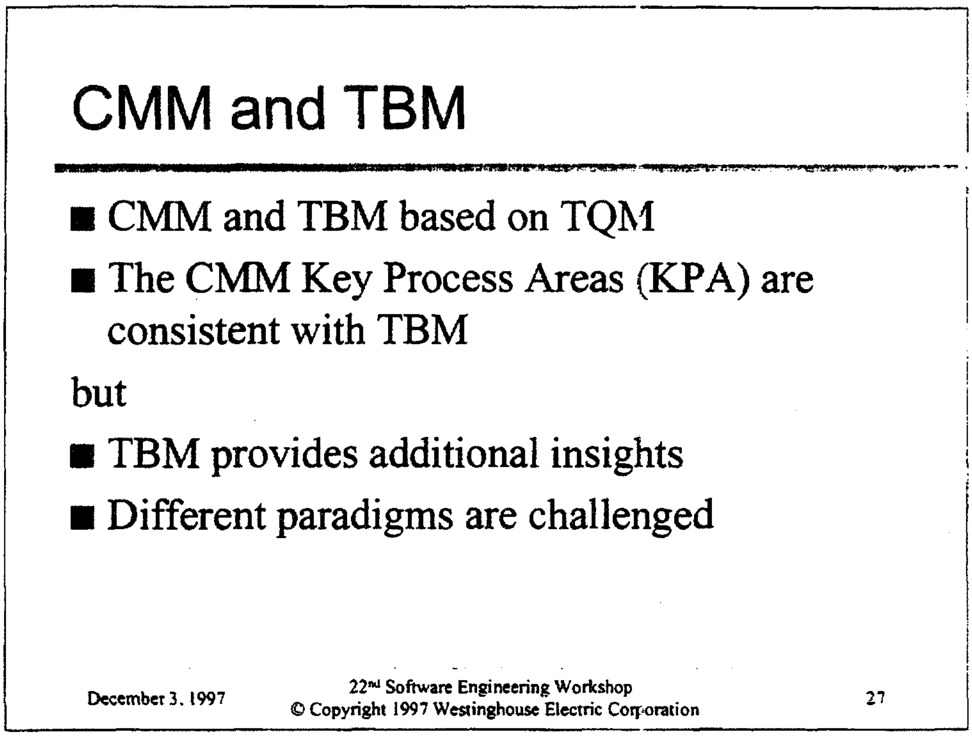
December 3, 1997 | 22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation | 25



Production of Defective Goods

- Any production of defects impacts cycle-time
- Passing on of defective goods increases the impact

December 3, 1997 | 22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation | 26



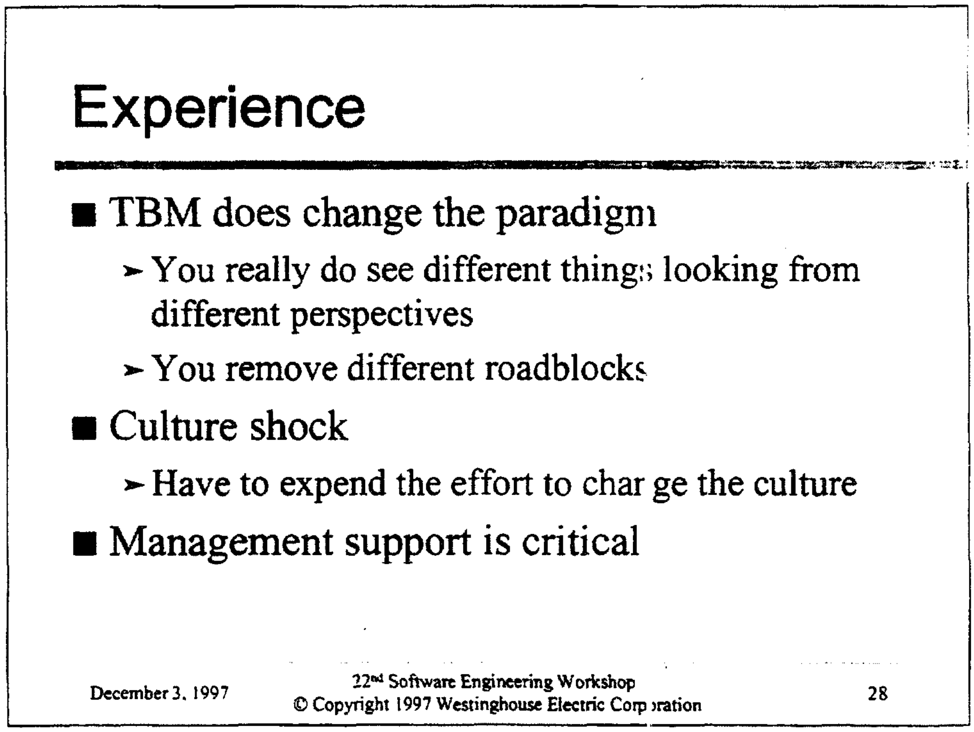
CMM and TBM

- CMM and TBM based on TQM
- The CMM Key Process Areas (KPA) are consistent with TBM

but

- TBM provides additional insights
- Different paradigms are challenged

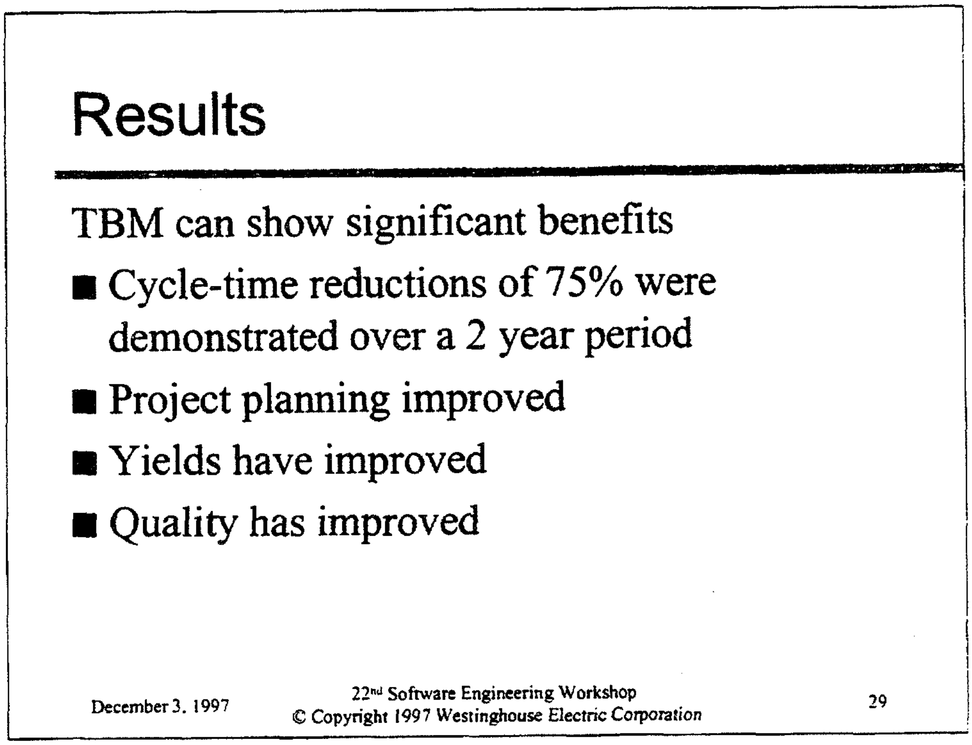
December 3, 1997 | 22nd Software Engineering Workshop | © Copyright 1997 Westinghouse Electric Corporation | 27



Experience

- TBM does change the paradigm
  - You really do see different things looking from different perspectives
  - You remove different roadblocks
- Culture shock
  - Have to expend the effort to char ge the culture
- Management support is critical

December 3, 1997 | 22nd Software Engineering Workshop | © Copyright 1997 Westinghouse Electric Corp›ration | 28



Results

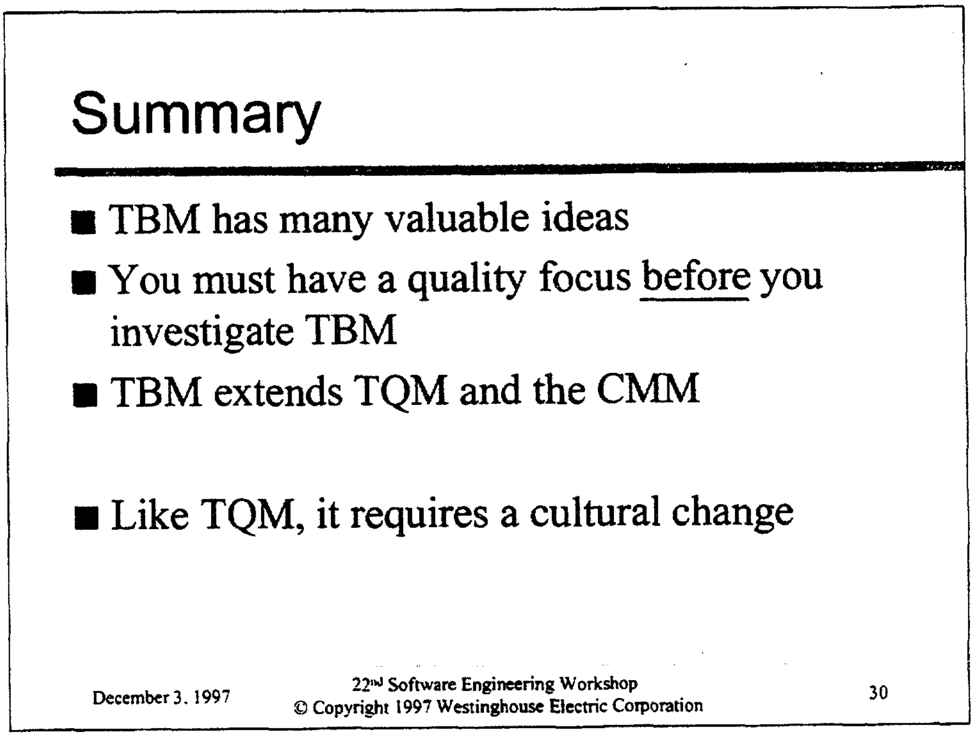
TBM can show significant benefits

- Cycle-time reductions of 75% were demonstrated over a 2 year period
- Project planning improved
- Yields have improved
- Quality has improved

December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

29



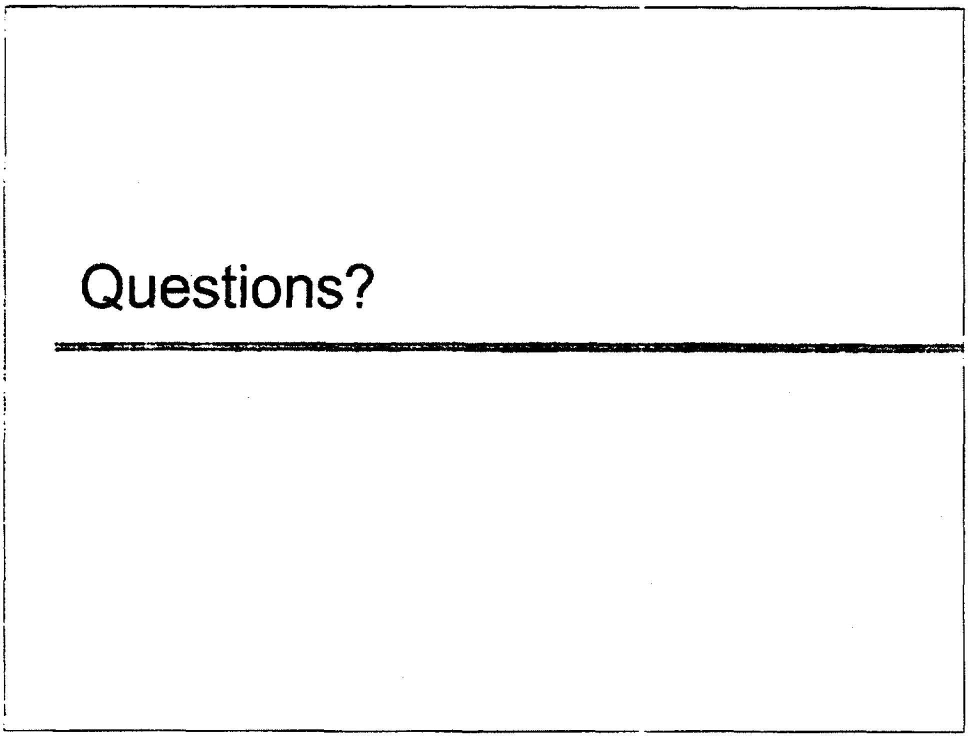
Summary

- TBM has many valuable ideas
- You must have a quality focus before you investigate TBM
- TBM extends TQM and the CMM
- Like TQM, it requires a cultural change

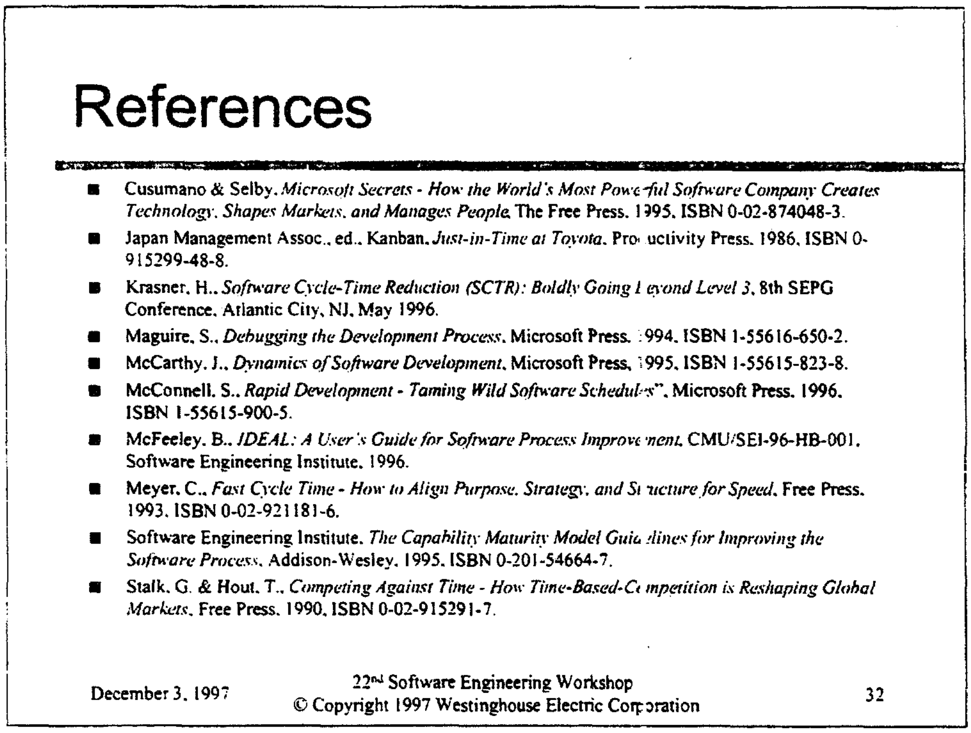
December 3, 1997

22nd Software Engineering Workshop
© Copyright 1997 Westinghouse Electric Corporation

30



Questions?



References

- Cusumano & Selby. *Microsoft Secrets - How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People.* The Free Press, 1995, ISBN 0-02-874048-3.
- Japan Management Assoc., ed., Kanban. *Just-in-Time at Toyota*. Productivity Press, 1986, ISBN 0-915299-48-8.
- Krasner, H., *Software Cycle-Time Reduction (SCTR): Boldly Going Beyond Level 3*, 8th SEPG Conference, Atlantic City, NJ, May 1996.
- Maguire, S., *Debugging the Development Process*, Microsoft Press, 1994, ISBN 1-55616-650-2.
- McCarthy, J., *Dynamics of Software Development*, Microsoft Press, 1995, ISBN 1-55615-823-8.
- McConnell, S., *Rapid Development - Taming Wild Software Schedules*", Microsoft Press, 1996, ISBN 1-55615-900-5.
- McFeeley, B., *IDEAL: A User's Guide for Software Process Improvement*, CMU/SEI-96-HB-001, Software Engineering Institute, 1996.
- Meyer, C., *Fast Cycle Time - How to Align Purpose, Strategy, and Structure for Speed*, Free Press, 1993, ISBN 0-02-921181-6.
- Software Engineering Institute, *The Capability Maturity Model Guidelines for Improving the Software Process*, Addison-Wesley, 1995, ISBN 0-201-54664-7.
- Stalk, G. & Hout, T., *Competing Against Time - How Time-Based-Competition is Reshaping Global Markets*, Free Press, 1990, ISBN 0-02-915291-7.

Session 4: Cost Models

Calibration Results of COCOMO II.1997
S. Devnani-Chulani, B. Boehm, and B. Clark, University of Southern California

Assessing the Accuracy of the COCOMO II.1997 Estimating Model
D. Reifer, Reifer Consultants, Inc.

Component-Based Software Development: Parameters Influencing Cost Estimation
R. Smith, A. Parrish, and J. Hale, University of Alabama

Calibrating the COCOMO II Post-Architecture Model

Sunita Devnani-Chulani
Bradford Clark
Barry Boehm
Center for Software Engineering
Computer Science Department
University of Southern California
Los Angeles, CA 90098-0781 USA
+1 213 740 6470
sdevnani@sunset.usc.edu
bkclark@sunset.usc.edu
boehm@sunset.usc.edu

ABSTRACT
The COCOMO II model was created to meet the need for a cost model that accounted for future software development practices. This resulted in the formulation of three submodels for cost estimation, one for composing applications, one for early lifecycle estimation and one for detailed estimation when the architecture of the product is understood. This paper describes the calibration procedures for the last model, Post-Architecture COCOMO II model, from eighty-three observations. The results of the multiple regression analysis and their implications are discussed. Future work includes further analysis of the Post-Architecture model, calibration of the other models, derivation of maintenance parameters, and refining the effort distribution for the model output.

Keywords
COCOMO II, cost estimation, metrics, multiple regression.

1 INTRODUCTION

The COCOMO II project started in July of 1994 with the intent to meet the projected need for a cost model that would be useful for the next generation of software development. The new model incorporated proven features of COCOMO 81 and Ada COCOMO models. COCOMO II has three submodels. The Application Composition model is used to estimate effort and schedule on projects that use Integrated Computer Aided Software Engineering tools for rapid application development. The Early Design and Post-Architecture models are used in estimating effort and schedule on software infrastructure, major applications, and embedded software projects.

The Early Design model is used when a rough estimate is needed based on incomplete project and product analysis. The Post-Architecture model is used when top level design is complete and detailed information is known about the project. Compared to COCOMO 81, COCOMO II added new cost drivers for application precedentedness, development flexibility, architecture and risk resolution, team cohesion, process maturity, required software reuse, documentation match to lifecycle needs, personnel continuity, and multi-site development. COCOMO II also eliminated COCOMO 81's concept of development modes and two COCOMO 81 cost drivers: turnaround time and modern programming practices.

This paper describes the calibration of the Post-Architecture model. The model determination process began with an expert Delphi process to determine a-priori values for the Post-Architecture model parameters. Data was collected on 112 projects over a two year period. From this dataset 83 projects were selected for use in model calibration. Projects with missing data or unexplainable anomalies were dropped. Model parameters that exhibited high correlation were consolidated. Multiple regression analysis was used to produce coefficients. These coefficients were used to adjust the previously assigned expert-determined model values. Stratification was used to improve model accuracy.

The t-values from the statistical analysis indicated that several of the variables' coefficients were not statistically significant. This appears to be due to lack of dispersion for some variables; imprecision of software effort, schedule, and cost driver data; and effects of partially correlated variables. As a result, the operational COCOMO II.1997 model uses a weighted average approach to determine an a-posteriori model as a percentage of the expert-determined a-priori cost-driver values and the data-determined values.

The resulting model produces estimates within 30% of the actuals 52% of the time for effort. If the model's multiplicative coefficient is calibrated to each of the major sources of project data, the resulting model produces estimates within 30% of the actuals 64% of the time for effort. It is therefore recommended that organizations using the model calibrate it using their own data. This increases model accuracy and produces a local optimum estimate for similar type projects.

Section 2 of this paper discusses the COCOMO II model that was calibrated. Section 3 describes the data used for calibration. Section 4 describes the calibration procedures, results, and future calibration strategy. Section 5 discusses the model forecast accuracy. Section 6 has our conclusions.

2 POST-ARCHITECTURE MODEL

The COCOMO II Post-Architecture model is fully described in [3 and 6]. The Post-Architecture model covers the actual development and maintenance of a software product. This stage of the lifecycle proceeds most cost-effectively if a software life-cycle architecture has been developed; validated with respect to the system's mission, concept of operation, and risk; and established as the framework for the product.

The Post-Architecture model predicts software development effort, Person Months (PM) , as shown in Equation 1. It has about the same granularity as the COCOMO 81 and Ada COCOMO models. It uses source instructions and / or function points for sizing, with modifiers for reuse and software breakage; a set of 17 multiplicative cost drivers (EM); and a set of 5 scaling cost drivers to determine the project's scaling exponent (SF). These scaling cost drivers replace the development modes (Organic, Semidetached, or Embedded) in the original COCOMO 81 model, and refine the four exponent-scaling factors in Ada COCOMO. The model has the form:

$$PM = A \cdot (Size)^{1.01+\sum_{j=1}^{5} SF_j} \cdot \prod_{i=1}^{17} EM_i \qquad (1)$$

All of the cost drivers are described in Table 1. Each driver can accept one of six possible ratings: Very Low (VL), Low (L), Nominal (N), High (H), Very High (VH), and Extra High (XH). Not all ratings are valid for all cost drivers. Table 2 shows the apriori values assigned to each rating before calibration.

The selection of scale factors (SF) in Equation 1 is based on the rationale that they are a significant source of exponential variation on a project's effort or productivity variation. Software cost estimation models often have an exponential factor to account for the relative economies or diseconomies of scale encountered in different size software projects.

The first five cost drivers in Table 1 are the five model scale factors. Each scale driver has a range of rating levels, from Very Low to Extra High. Each rating level has a specific value as shown in Table 2. A project's scale factors, SF_{1-5}, are summed and used to determine a scale exponent. The scale factor constant, 1.01, is set to a value greater than one because it is believed that scale factors exhibit diseconomies of scale. This is generally due to two main factors: growth of interpersonal communications overhead and growth of large-system integration overhead. Larger projects will have more personnel, and thus more interpersonal communications paths consuming overhead. Integrating a small product as part of a larger product requires not only the effort to develop the small product, but also the additional overhead effort to design, maintain, integrate, and test its interfaces with the remainder of the product. See [1] for a further discussion of software economies and diseconomies of scale.

Recent research has confirmed diseconomies of scale influence on effort with the Process Maturity (PMAT) scaling cost driver [4]. An econometric log-log model and the COCOMO II Post-Architecture model were used to determine effect on effort. The log-log model employed PMAT as a multiplicative cost driver and the COCOMO II Post-Architecture model used PMAT as shown in Equation 1. The multiple regression results showed that PMAT was more significant when used as a scaling cost driver. For projects in the 30 - 120 KSLOC range, the analysis indicated that a one level improvement in Process Maturity corresponded to a 15 - 21% reduction in effort, after normalization for the effects of other cost drivers.

Table 1. COCOMO II Cost Drivers

Abr.	Name and Description
PREC	Precendentedness: Captures the organization's understanding of product objectives and required technology.
FLEX	Development Flexibility: Expresses the degree of conformance to software requirements and external interface standards.
RESL	Architecture and Risk Resolution: Rates the integrity and understanding of the product software architecture and the number / criticality of unresolved risk items.
TEAM	Team cohesion: Captures the consistency of stakeholder objectives and the willingness of all parties to work together as a team.
PMAT	Process Maturity: Maturity of the software process used to produce the product; based on the Software Capability Maturity Model

Abr.	Name and Description
RELY	Required Software Reliability: Degree of assurance to which the software will perform its intended function.
DATA	Data Base Size: Captures the effect that large data requirements have on product development.
CPLX	Product Complexity: Characterizes the product's complexity in five areas; control operations, computational operations, device-dependent operations, data management operations, and user interface management operations
RUSE	Required Reusability: Accounts for the additional effort needed to construct components intended for reuse on the current or future projects.
DOCU	Documentation Match to Life-cycle Needs: Evaluated in terms of the suitability of the project's documentation to its life-cycle needs
TIME	Time Constraint: Measure of the execution time constraint imposed upon a software system.
STOR	Storage Constraint: Degree of main storage constraint imposed on a software system.
PVOL	Platform Volatility: Volatility of the complex of hardware and software (OS, DBMS, etc.) the software product calls on to perform its tasks.
ACAP	Analyst Capability: Captures analysis and design ability, efficiency and thoroughness, and the ability to communicate and cooperate.
PCAP	Programmer Capability: Captures ability, efficiency and thoroughness, and the ability to communicate and cooperate. Considers programmers as a team rather than as individuals.
AEXP	Applications Experience: Rates the level of applications experience of the project team.
PEXP	Platform Experience: Experience in using the complex of hardware and software (OS, DBMS, etc.) the software product calls on to perform its tasks.
LTEX	Language and Tool Experience: Measure of the level of programming language and software tool experience of the project team.
PCON	Personnel Continuity: Project's annual personnel turnover.
TOOL	Use of Software Tools: Captures the productivity impact of tools ranging from simple edit and code tools to integrated, pro-active lifecycle support tools.
SITE	Multi-Site Development: Assesses site collocation (from fully collocated to international distribution) and communication support (from surface mail and some phone access to full interactive multimedia).
SCED	Required Development Schedule: Measures the schedule constraint imposed on the project team

Table 2 shows the apriori values assigned to the scaling and multiplicative cost drivers. An example of the criteria for selecting a rating for a driver is given for the Required Reusability (RUSE) cost driver in Table 4. All cost drivers are qualitative, except for size, and are measured by selecting one of six ratings: Very Low (VL), Low (L), Nominal (N), High (H), Very High (VH), and Extra High (XH). As can be seen in Table 2 not all six rating levels were valid for all cost drivers. An example of this is the lack of a VL rating for the RUSE cost driver (see Table 4).

Table 2. Apriori Model Values

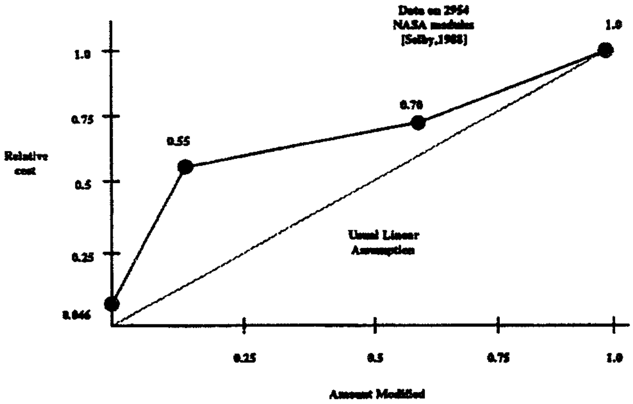
Driver	Sym	VL	L	N	H	VH	XH
PREC	SF_1	0.05	0.04	0.03	0.02	0.01	0.0
FLEX	SF_2	0.05	0.04	0.03	0.02	0.01	0.0
RESL	SF_3	0.05	0.04	0.03	0.02	0.01	0.0
TEAM	SF_4	0.05	0.04	0.03	0.02	0.01	0.0
PMAT	SF_5	0.05	0.04	0.03	0.02	0.01	0.0
RELY	EM_1	0.75	0.88	1.00	1.15	1.40	
DATA	EM_2		0.94	1.00	1.08	1.16	
CPLX	EM_3	0.75	0.88	1.00	1.15	1.30	1.65
RUSE	EM_4		0.89	1.00	1.16	1.34	1.56
DOCU	EM_5	0.85	0.93	1.00	1.08	1.17	
TIME	EM_6			1.00	1.11	1.30	1.66
STOR	EM_7			1.00	1.06	1.21	1.56
PVOL	EM_8		0.87	1.00	1.15	1.30	
ACAP	EM_9	1.5	1.22	1.00	0.83	0.67	
PCAP	EM_{10}	1.37	1.16	1.00	0.87	0.74	
PCON	EM_{11}	1.26	1.11	1.00	0.91	0.83	
AEXP	EM_{12}	1.23	1.10	1.00	0.88	0.80	
PEXP	EM_{13}	1.26	1.12	1.00	0.88	0.80	
LTEX	EM_{14}	1.24	1.11	1.00	0.9	0.82	
TOOL	EM_{15}	1.20	1.10	1.00	0.88	0.75	
SITE	EM_{16}	1.24	1.10	1.00	0.92	0.85	0.79
SCED	EM_{17}	1.23	1.08	1.00	1.04	1.10	

The Size input to the Post-Architecture model, Equation 1, includes adjustments for breakage effects, adaptation, and reuse. Size can be expressed as Unadjusted Function Points (UFP) [9] or thousands of source lines of code (KSLOC).

The COCOMO II size reuse model is nonlinear and is based on research done by Selby [12]. Selby's analysis of reuse costs across nearly 3000 reused modules in the NASA Software Engineering Laboratory indicates that the reuse cost function is nonlinear in two significant ways (see Figure 1):

- It does not go through the origin. There is generally a cost of about 5% for assessing, selecting, and assimilating the reusable component.
- Small modifications generate disproportionately large costs. This is primarily due to two factors: the cost of understanding the software to be modified, and the relative cost of interface checking.

Figure 1. Non-linear Effects of Reuse



The COCOMO II sizing model captures this non-linear effect with six parameters: percentage of design modified (DM); the percentage of code modified (CM); the percentage of modification to the original integration effort required for integrating the reused software (IM); software understanding (SU) for structure, clarity, and self-descriptiveness; unfamiliarity with the software (UNFM) for programmer knowledge of the reused code; and assessment and assimilation (AA) for fit of the reused module to the application.

3 DATA COLLECTION

Data collection began in September 1994. The data came from organizations that were Affiliates of the Center for Software Engineering at the University of Southern California and some other sources. These organizations represent the Commercial, Aerospace, and Federally Funded Research and Development Centers (FFRDC) sectors of software development with Aerospace being most represented in the data.

Data was recorded on a data collection form that asked between 33 and 59 questions depending on the degree of source code reuse [7]. The data collected was historical, i.e. observations were completed projects. The data was collected either by site visits, phone interviews, or by contributors sending in completed forms. As a baseline for the calibration database, some of the COCOMO 1981 projects and Ada COCOMO projects were converted to COCOMO II data inputs. The total observations used in the calibration was 83, coming from 18 different organizations. This dataset formed the basis for an initial calibration.

All data that was collected was labeled with a generic identifier. The source of the data was stored in a different location to protect its confidentiality. The data was stored in a locked room with access restricted to project research personnel. When the data was entered into the repository, it was checked for completeness and consistency. Incomplete or inconsistent data points were dropped from the calibration dataset.

A frequent question is what defines a line of source code. Appendix B in the Model Definition Manual [6] defines a logical line of code using a framework described in [10]. However the data collected to date has exhibited local variations in interpretation of counting rules, which is one of the reasons that local calibration produces more accurate model results. Local calibration results are discussed later in this paper.

The data collected included the actual effort and schedule spent on a project. Effort is in units of Person Months. A person month is 152 hours a month and includes development and management hours. Schedule is calendar months. Adjusted KSLOC is the thousands of lines of source code count adjusted for breakage and reuse. The following three histograms show the frequency of responses for this data.

Overall, the 83 data points ranged in size from 2 to 1,300 KSLOC, in effort from 6 to 11,400 person months, and in schedule from 4 to 180 months.

Figure 2. Data Distribution for Person Months

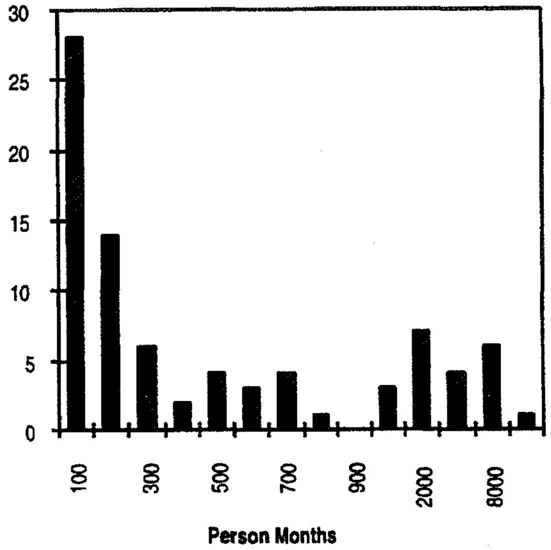


Figure 3. Data Distribution for Schedule

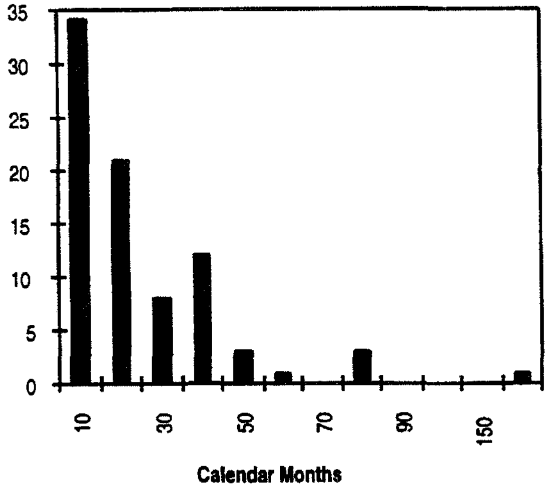
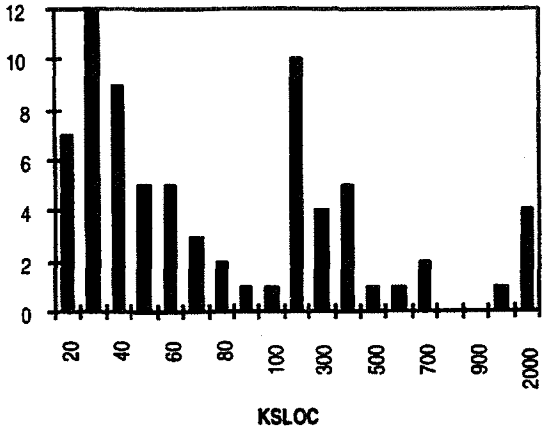


Figure 4. Data Distribution for Size



4. MODEL CALIBRATION

The comprehensive calibration method adjusts all cost driver parameters in the Post-Architecture model. This level of analysis requires a lot of data; usually more than an organization has available. We plan to annually update the model parameter values based on newly added and existing data in our repository. Hence the COCOMO II model name will have a year date after it identifying the set of values on which the model is based, e.g. COCOMO II.1997.

The statistical method we used to calibrate the model is multiple regression analysis. This analysis finds the least squares error solution between the model parameters and the data. However, multiple regression analysis, which derives the values for b_i, can only be applied to linear models: $b_0 + b_1A + b_2B + b_3C = D$. The COCOMO model as shown in Equation 2 is a non-linear model. This means that there are values to be derived in the exponent portion of the model. To solve this problem we transform the non-linear model in Equation 2 into a linear model with three steps.

The first step is to expand Equation 1 into unique factors. Each factor will be calibrated.

$$PM = A \cdot (Size)^{1.01} \cdot (Size)^{SF_1} \cdot\cdots\cdot (Size)^{SF_5} \cdot EM_1 \cdot EM_2 \cdots EM_{16} \cdot EM_{17} \quad (2)$$

The second step is to heuristically set the values of the exponential and multiplicative cost drivers. These values are the apriori values shown earlier in Table 2. The third step is to transform both sides of the multiplicative form of Equation 2 into a linear form using logarithms to the base *e*. This technique is called a log-log transformation [13].

$$\begin{aligned}\ln(PM \div Size^{1.01}) = B_0 + B_1 \cdot SF_1 \cdot \ln(Size) + \\ B_2 \cdot SF_2 \cdot \ln(Size) + \cdots + \\ B_5 \cdot SF_5 \cdot \ln(Size) + \\ B_6 \cdot \ln(EM_1) + \\ B_7 \cdot \ln(EM_2) + \cdots + \\ B_{21} \cdot \ln(EM_{16}) + \\ B_{22} \cdot \ln(EM_{17})\end{aligned} \qquad (3)$$

Multiple regression analysis is performed on the linear model in log space. The derived coefficients, B_i, from regression analysis are used to adjust the apriori values. The log-log transformation is supported by analysis of the data which shows other transformations such as log-linear, square root-linear, and quad root-linear resulted in non-constant variance errors. Only the log-log transform satisfies the test for independence between errors and the observations. Equation 3 shows the fixed exponent, $Size^{1.01}$, removed from the analysis as we want the scale factors to explain as much of the variance as possible.

For calibration purposes some cost drivers in the original definition of the model were aggregated into new cost drivers because of the high correlation between them. These high correlations introduce numerical instabilities into the data analysis. The Analyst Capability and Programmer Capability were combined to form the Personnel Capability (PERS) cost driver. Execution Time Constraint and Main Storage Constraint were combined to form Resource Constraints (RCON): EM_{14} and EM_{15} in Table 3.

4.1. Results of Effort Calibration

There were 83 observations used in the multiple regression analysis. Of those observations, 59 were used to create a baseline set of coefficients. The response variable was Person Months (PM). The predictor variables were size (adjusted for reuse and breakage) and all of the cost drivers described in Table 1. The coefficients obtained from the analysis are shown in Table 3. These coefficients are applied to the model using Equation 4. The constant, A, is derived from raising e to the coefficient, B_0.

$$\begin{aligned}PM = e^{B_0} \cdot (Size)^{1.01} \cdot (Size)^{B_1 \cdot SF_1} \cdot \\ (Size)^{B_2 \cdot SF_2} \cdot \ldots \cdot (Size)^{B_5 \cdot SF_5} \cdot \\ EM_1^{B_6} \cdot EM_2^{B_7} \cdot \ldots \cdot \\ EM_{14}^{B_{19}} \cdot EM_{15}^{B_{20}}\end{aligned} \qquad (4)$$

Table 3. Estimated Coefficients

Sym	Coefficient	Estimate	t-value
A	B_0	0.70188	3.026
PREC	B_1 for SF_1	-0.90196	-0.621
FLEX	B_2 for SF_2	3.14218	2.074
RESL	B_3 for SF_3	-0.55861	-0.293
TEAM	B_4 for SF_4	0.86614	0.509
PMAT	B_5 for SF_5	0.08844	0.068
RELY	B_6 for EM_1	0.79881	1.511
DATA	B_7 for EM_2	2.52797	3.493
RUSE	B_8 for EM_3	-0.44410	-0.913
DOCU	B_9 for EM_4	-1.32819	-1.999
CPLX	B_{10} for EM_5	1.13191	2.605
PVOL	B_{11} for EM_6	0.85830	1.612
AEXP	B_{12} for EM_7	0.56053	0.920
PEXP	B_{13} for EM_8	0.69690	1.321
LTEX	B_{14} for EM_9	-0.04214	-0.063
PCON	B_{15} for EM_{10}	0.30826	0.260
TOOL	B_{16} for EM_{11}	2.49512	2.243
SITE	B_{17} for EM_{12}	1.39701	1.679
SCED	B_{18} for EM_{13}	2.84075	3.670
PERS	B_{19} for EM_{14}	0.98747	4.282
RCON	B_{20} for EM_{15}	1.36588	5.001

The negative coefficient estimates do not support the ratings for which the data was gathered. To see the effect of a negative coefficient, Table 4 gives the ratings, apriori values, and fully adjusted values for RUSE. Based on the definition of the Required Reusability cost driver, RUSE, the apriori model values indicate that as the rating increases from Low (L) to Extra High (XH), the amount of required effort will also increase. This rationale is consistent with the results of 12 studies of the relative cost of writing for reuse compiled in [11]. The adjusted values determined from the data sample indicate that as more software is built for wider ranging reuse less effort is required. As shown in Figure 5, diamonds versus triangles, this is inconsistent with the expert-determined multiplier values obtained via the COCOMO II Affiliate representatives.

Table 4. RUSE Cost Driver

		Values	
RUSE	Definition	Apriori	Adjusted
L	None	0.89	1.05
N	Across project	1.00	1.00
H	Across program	1.16	0.94
VH	Across product line	1.34	0.88
XH	Across multiple product lines	1.56	0.82

A possible explanation for the phenomenon is the frequency distribution of the data used to calibrate RUSE, Figure 6. There were a lot of responses that were "I don't know" or "It does not apply." These are essentially entered as a Nominal rating in the model. This weakens the data analysis in two ways: via weak dispersion of rating values, and via possibly inaccurate data values.

Figure 5. RUSE Calibrated Values

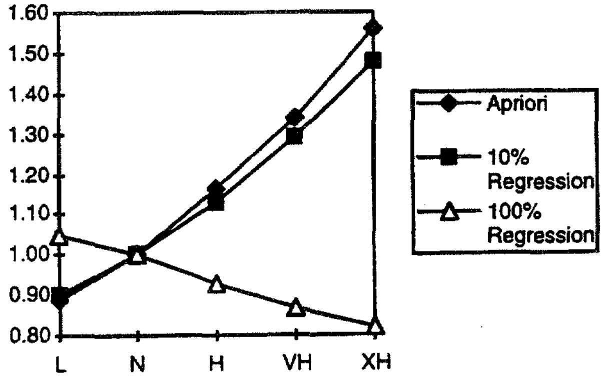
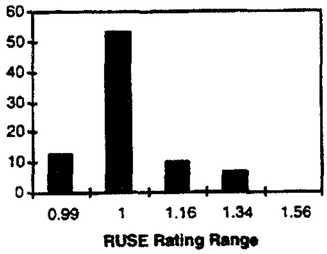


Figure 6. Frequency of RUSE



The COCOMO II Affiliate users were reluctant to use a pure regression-based model with counterintuitive trends such as the triangles in Figure 5 for RUSE. We therefore used a weighted average approach to determine an aposteriori set of cost driver values as a weighted average of the apriori cost drivers and the regression-determined cost drivers. Using 10% of the data-driven and 90% of the apriori values, Table 5 shows the COCOMO II.1997 calibrated values. The constant, A, is set to 2.45.

The 10% weighting factor was selected after comparison runs using other weighing factors were found to produce less accurate results. This moves the model parameters in the direction suggested by the regression coefficients but retains the rationale contained within the apriori values. As more data is used to calibrate the model, a greater percentage of the weight will be given to the regression determined values. Thus the strategy is to release annual updates to the calibrated parameters with each succeeding update producing more data driven parameter values.

Table 5. COCOMO II.1997 Cost Driver Values

Driver	Sym	VL	L	N	H	VH	XH
PREC	SF_1	0.0405	0.0324	0.0243	0.0162	0.0081	0.00
FLEX	SF_2	0.0607	0.0486	0.0364	0.0243	0.0121	0.00
RESL	SF_3	0.0422	0.0338	0.0253	0.0169	0.0084	0.00
TEAM	SF_4	0.0494	0.0395	0.0297	0.0198	0.0099	0.00
PMAT	SF_5	0.0454	0.0364	0.0273	0.0182	0.0091	0.00
RELY	EM_1	0.75	0.88	1.00	1.15	1.39	
DATA	EM_2		0.93	1.00	1.09	1.19	
CPLX	EM_3	0.75	0.88	1.00	1.15	1.30	1.66
RUSE	EM_4		0.91	1.00	1.14	1.29	1.49
DOCU	EM_5	0.89	0.95	1.00	1.06	1.13	
TIME	EM_6			1.00	1.11	1.31	1.67
STOR	EM_7			1.00	1.06	1.21	1.57
PVOL	EM_8		0.87	1.00	1.15	1.30	
ACAP	EM_9	1.50	1.22	1.00	0.83	0.67	
PCAP	EM_{10}	1.37	1.16	1.00	0.87	0.74	
PCON	EM_{11}	1.24	1.10	1.00	0.92	0.84	
AEXP	EM_{12}	1.22	1.10	1.00	0.89	0.81	
PEXP	EM_{13}	1.25	1.12	1.00	0.88	0.81	
LTEX	EM_{14}	1.22	1.10	1.00	0.91	0.84	
TOOL	EM_{15}	1.24	1.12	1.00	0.86	0.72	
SITE	EM_{16}	1.25	1.10	1.00	0.92	0.84	0.78
SCED	EM_{17}	1.29	1.10	1.00	1.00	1.00	

The research on the Process Maturity (PMAT) cost driver has shown the data-driven approach to be a credible. A larger dataset was used to determine PMAT's influence on effort. PMAT was statistically significant with the large dataset compared to the unknown significance shown in Table 3 (the t-value is less than 1.96) [4]. This was caused by the additional data having a wider dispersion of responses for PMAT across its rating criteria.

4.2 Results of Schedule Calibration

COCOMO II provides an estimation of schedule (TDEV), Equation 5. Only the constant for the schedule equation was calibrated. The calibrated schedule constant, A, is set to 2.66. The apriori value was 3.0.

$$TDEV = [A \cdot \left(\overline{PM}\right)^{(0.33+0.2\cdot\sum SF_i)}] \cdot \frac{SCED\%}{100} \tag{5}$$

$\overline{PM}$ is the estimation of effort without the *SCED* cost driver included. *SF* are the scaling cost drivers earlier.

5 MODEL ACCURACY

5.1 Criteria

Four criteria are used to judge how well the model forecasts effort and schedule. These criteria are the Adjusted R^2 (Adj-R^2), estimated Standard Deviation ($\hat{\sigma}$), and Prediction at level L (PRED(L)). Adjusted R^2 is the coefficient of determination, R^2, adjusted for the number of cost drivers and the number of observations [8]. R^2 reports the proportion of variation explained by the model.

The estimated Standard Deviation is an indication of the variation in the error of the prediction. It is based on the difference between the known effort to complete a software project and the model estimated effort.

PRED(L) is the percentage of estimated values that were within L percent of the actual values [5]. For example, PRED(.25) reports the percentage of estimates that were within 25% of the actual values. This measure depends on the measurement of error. The error measurement used in our results is called Proportional Error.

Proportional Error (PE) is a measure of relative error used to compute PRED(L). As the estimates become larger for larger projects, the difference between the known effort to complete a software project and the model estimated effort is normalized for project size. Equation 6 shows this normalization.

$$PE = \begin{cases} [PM_{est} + PM_{act}] - 1, (PM_{est} - PM_{act}) \geq 0 \\ -[PM_{act} + PM_{est}] + 1, (PM_{est} - PM_{act}) < 0 \end{cases} \tag{6}$$

5.2 Results

With all 83 observations, the 10% adjusted values for the COCOMO II parameters produced the effort forecast measurements shown in Table 6. There are two columns, Before Stratification and After Stratification. The first column shows accuracy results using the same constant, A, and cost drivers for all observations. The second column shows the accuracy results using a different constant, A, and different fixed exponent for each different organization that contributed data. The results are better for the stratified model. The Standard Deviation ($\hat{\sigma}$) for the actual effort values was 1953 Person Months. It can be seen that the $\hat{\sigma}$ for the estimated values in the stratified model is better than the $\hat{\sigma}$ for the actual values.

Table 6. Effort Forecast Results

Effort Prediction	Before Stratification	After Stratification
Adj_R^2	0.93	0.95
$\hat{\sigma}$	2165	1925
PRED(.20)	42%	47%
PRED(.25)	47%	56%
PRED(.30)	51%	66%

These results are not as good as the COCOMO 81 model. This we believe is due to the small amount of data available for model calibration. However, it can be seen that calibration of the model to local conditions increases accuracy.

6 CONCLUSIONS

Data collection is still on going. The first calibrated release of the model occurred early in 1997. Until we can produce 100% data driven values, COCOMO II will under go annual updates to all of its cost drivers using the methods described in this paper. Each release will increasingly depend on data-driven values.

Stratification produced very good results. As with the previous COCOMO models, calibration of the constant, A, and the fixed exponent, 1.01, to local conditions is highly recommended. This feature is available in a commercial implementation of COCOMO II, COSTAR's Calico tool, and is also available in the free software tool, USC COCOMO II 1997.1.

In our data collection, there is now an entry for "I don't know." This reflects the paradox in attempting to create a cost model for the future using data from the past. Cost drivers such as RUSE reflect practices that were not emphasized in historical project data.

Future Work

With more data we are going to make a through examination of the cost drivers that have a negative coefficient The theory upon which the model is based does not support negative coefficients. Our experience with the Process Maturity cost driver has demonstrated the positive effects of using a larger dataset. There is also a need to calibrate cost drivers for their influence during maintenance. COCOMO 81 had this feature. The distribution of effort across lifecycle phases needs to be updated. This is challenging because of different lifecycle models that are used today, e.g. spiral, iterative, evolutionary. COCOMO II's sizing model is very comprehensive but there was not enough data to fully check its validity. Additionally the relationship between Unadjusted Function Points and logical / physical source lines of code needs further study.

ACKNOWLEDGMENTS

Data, expertise, and funding support from the 27 COCOMO II Affiliates: Aerospace Corporation, Air Force Cost Analysis Agency, Allied Signal, Bellcore, The Boeing Co., E-Systems/Raytheon, Electronic Data Systems Corporation, Federal Aviation Administration, Hughes Aircraft Company, Institute for Defense Analysis, Jet Propulsion Laboratory, Litton Data Systems, Lockheed/Martin, Lucent Technologies, Motorola Inc., Network Programs, Inc., Northrop Grumman Corporation, Rational Software Corporation, Science Applications International Corporation, Software Engineering Institute (CMU), Software Productivity Consortium, Sun Microsystems Inc., Texas Instruments, TRW, Inc., U.S. Air Force Rome Laboratory, U.S. Army Research Laboratory, and Xerox Corporation.

REFERENCES

1. Banker, R., H. Change, and C. Kemerer, "Evidence on Economies of Scale in Software Development," Information and Software Technology, vol. 36, no. 5, 1994, pp.275-828.

2 . Boehm, B., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

3. Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy, R. Selby, "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," Annals of Software Engineering Special Volume on Software Process and Product Measurement, J.D. Arthur and S.M. Henry (Eds.), J.C. Baltzer AG, Science Publishers, Amsterdam, The Netherlands, Vol 1, 1995, pp. 45 - 60.

4. Clark, B., "The Effects of Process Maturity on Software Development Effort," Ph.D. Dissertation, Computer Science Department, University of Southern California, Aug. 1997.

5. Conte, S., H. Dunsmore, and V. Shen, Software Engineering Metrics and Models, Benjamin/Cummings, Menlo Park, Ca., 1986.

6. Center for Software Engineering , "COCOMO II Model Definition Manual," Computer Science Department, University of Southern California, Los Angeles, Ca. 90089, http://sunset.usc.edu/Cocomo.html, 1997.

7. Center for Software Engineering, "COCOMO II Cost Estimation Questionnaire," Computer Science Department, University of Southern California, Los Angeles, Ca. 90089, http://sunset.usc.edu/Cocomo.html, 1997.

8. Griffiths, W., R. Hill, and G. Judge, Learning and Practicing Econometrics, John Wiley & Sons, Inc., New York, N.Y., 1993.

9. IFPUG, Function Point Counting Practices: Manual Release 4.0, International Function Point User's Group, 1994.

10.Park. R., "Software Size Measurement: A Framework for Counting Source Statements," CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, Pa., 1992.

11.Poulin, J., Measuring Software Reuse, Addison-Wesley, Reading, Ma., 1997.

12.Selby, R., "Empirically Analyzing Software Reuse in a Production Environment," in Software Reuse: Emerging Technology, W. Tracz (Ed.), IEEE Computer Society Press, 1988, pp.176-189.

13.Weisberg, S., Applied Linear Regression, 2nd Ed., John Wiley and Sons, New York, N.Y., 1985.

22nd SEW COVER PAGE

TITLE OF PAPER:

Calibrating the COCOMOII post Architecture Model

CONTACT PERSON:

Name: Sunita Devnani Chulani

Affiliation : Center for Software Engineering, University Of Southern California

Street : Dept Of Computer Science

Address: University Of Southern California

City : Los Angeles

State/Province : CA

Postal/Zip Code : 90089 -0781

Country: US

Email: sdevnani@sunset.usc.edu

Telephone : 213 740 4927

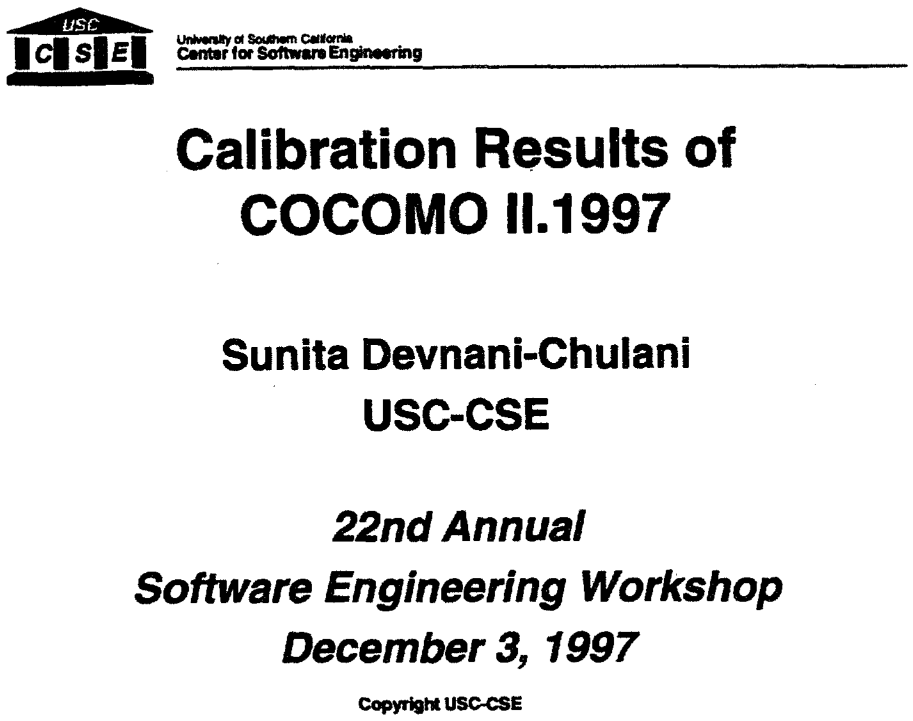
Additional Authors

Name : Brad Clark

Affiliation : Center for Software Engineering, University Of Southern California

Name : Barry Boehm

Affiliation : Center for Software Engineering, University Of Southern California



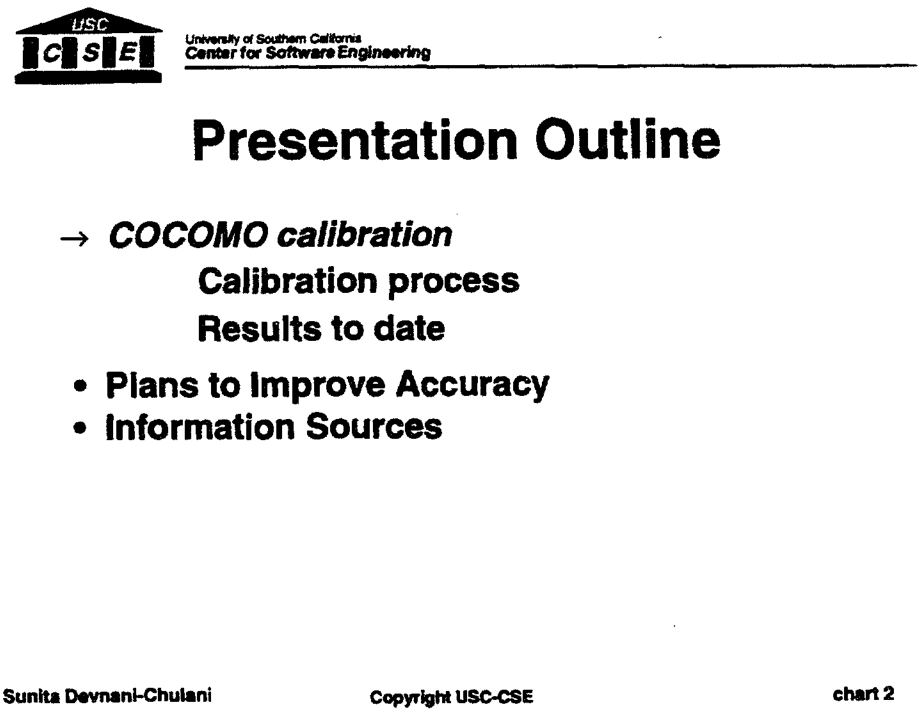
University of Southern California
Center for Software Engineering

Calibration Results of COCOMO II.1997

Sunita Devnani-Chulani
USC-CSE

22nd Annual
Software Engineering Workshop
December 3, 1997

Copyright USC-CSE



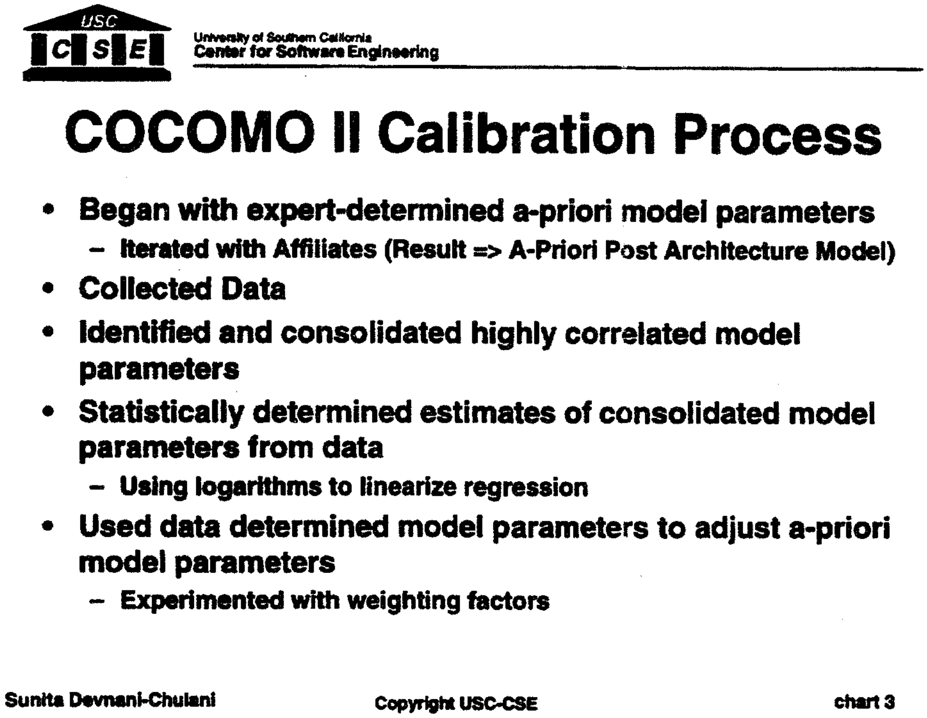
University of Southern California
Center for Software Engineering

Presentation Outline

→ ***COCOMO calibration***
  - **Calibration process**
  - **Results to date**

- **Plans to Improve Accuracy**
- **Information Sources**

Sunita Devnani-Chulani Copyright USC-CSE chart 2

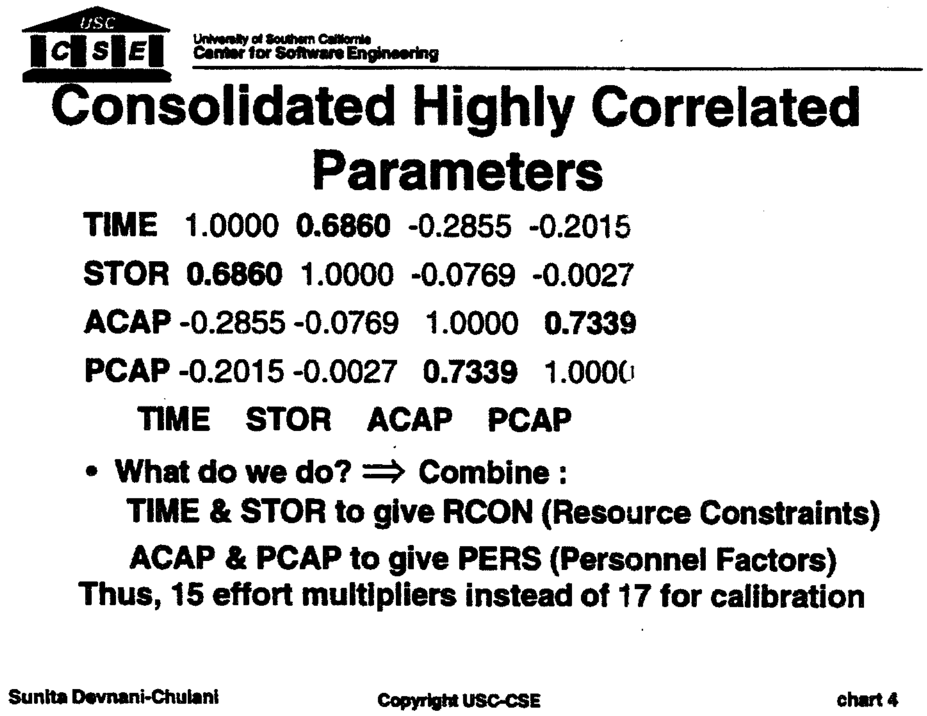


USC CSE University of Southern California Center for Software Engineering

COCOMO II Calibration Process

- **Began with expert-determined a-priori model parameters**
  - **Iterated with Affiliates (Result => A-Priori Post Architecture Model)**
- **Collected Data**
- **Identified and consolidated highly correlated model parameters**
- **Statistically determined estimates of consolidated model parameters from data**
  - **Using logarithms to linearize regression**
- **Used data determined model parameters to adjust a-priori model parameters**
  - **Experimented with weighting factors**

Sunita Devnani-Chulani Copyright USC-CSE chart 3



USC CSE University of Southern California Center for Software Engineering

Consolidated Highly Correlated Parameters

TIME	1.0000	**0.6860**	-0.2855	-0.2015
STOR	**0.6860**	1.0000	-0.0769	-0.0027
ACAP	-0.2855	-0.0769	1.0000	**0.7339**
PCAP	-0.2015	-0.0027	**0.7339**	1.0000
	TIME	**STOR**	**ACAP**	**PCAP**

- **What do we do? ⟹ Combine :**
  TIME & STOR to give RCON (Resource Constraints)
  ACAP & PCAP to give PERS (Personnel Factors)

Thus, 15 effort multipliers instead of 17 for calibration

Sunita Devnani-Chulani Copyright USC-CSE chart 4

University of Southern California
Center for Software Engineering

Statistical Data Analysis

Variable	Minimum	Maximum	Ratio (Max/Min)
EFFORT	6	11400	1900
SIZE	2.6	1292.8	497

Thus, we took log transforms to normalize the response variable.
Also, we took log transforms to linearize the parametrized model.

Sunita Devnani-Chulani Copyright USC-CSE chart 5

University of Southern California
Center for Software Engineering

Expanded COCOMO

- **Distributed the Scale Factors**
- **Resulted in 21 predictor variables i.e. 15 Effort Multipliers + 5 Scale Factors + $(\text{Size})^{1.01}$**

$$PM_{est} = A \cdot (Size)^{1.01} \cdot (Size)^{SF_1} \cdot (Size)^{SF_2} \cdots EM_1 \cdots EM_{15}$$

Log Transformed COCOMO:

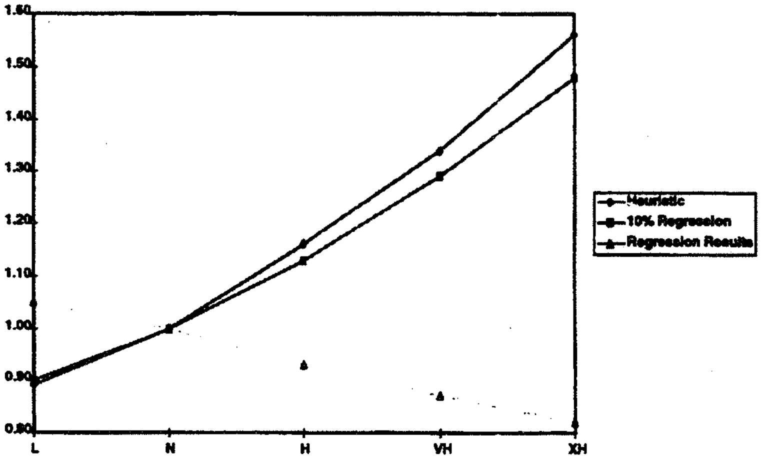
$$\ln(PM_{est}) - \ln(Size)\text{^}1.01 = \ln(A) + SF_1 \ln(Size) + \cdots + \ln(EM_{15})$$

- **Regression analysis derived the coefficients, B_i, for each factor**

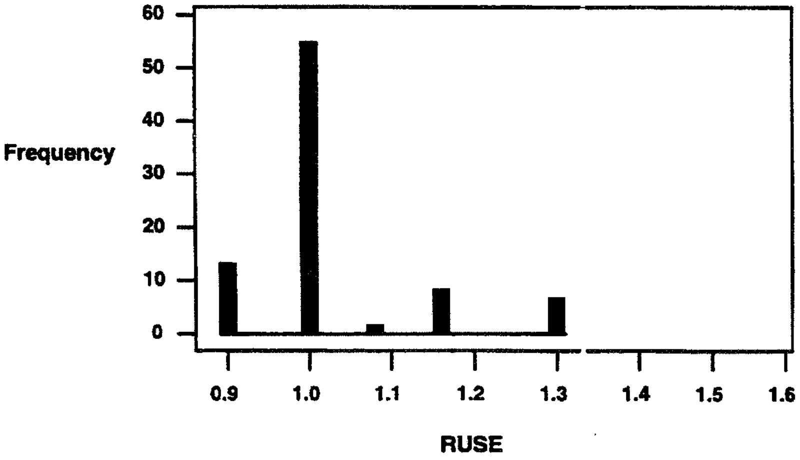
Sunita Devnani-Chulani Copyright USC-CSE chart 6

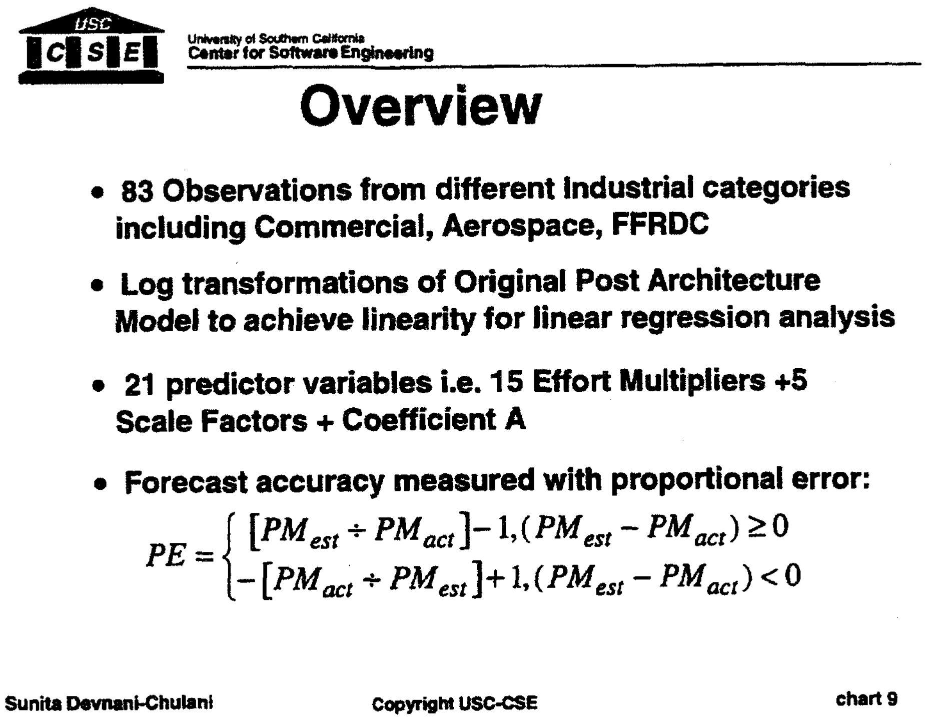
RUSE Effort Multiplier

- **Example of the effect of a negative coefficient**



Distribution of RUSE





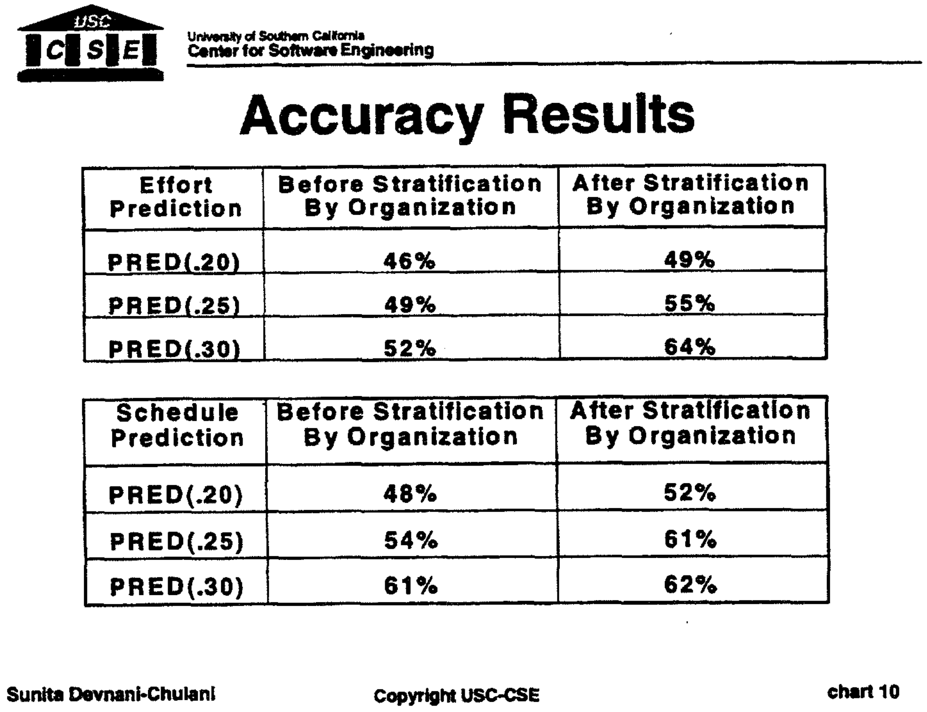
University of Southern California
Center for Software Engineering

Overview

- 83 Observations from different Industrial categories including Commercial, Aerospace, FFRDC
- Log transformations of Original Post Architecture Model to achieve linearity for linear regression analysis
- 21 predictor variables i.e. 15 Effort Multipliers +5 Scale Factors + Coefficient A
- Forecast accuracy measured with proportional error:

$$PE = \begin{cases} [PM_{est} \div PM_{act}] - 1, (PM_{est} - PM_{act}) \geq 0 \\ -[PM_{act} \div PM_{est}] + 1, (PM_{est} - PM_{act}) < 0 \end{cases}$$

Sunita Devnani-Chulani Copyright USC-CSE chart 9



University of Southern California
Center for Software Engineering

Accuracy Results

Effort Prediction	Before Stratification By Organization	After Stratification By Organization
PRED(.20)	46%	49%
PRED(.25)	49%	55%
PRED(.30)	52%	64%

Schedule Prediction	Before Stratification By Organization	After Stratification By Organization
PRED(.20)	48%	52%
PRED(.25)	54%	61%
PRED(.30)	61%	62%

Sunita Devnani-Chulani Copyright USC-CSE chart 10

USC CSE University of Southern California
Center for Software Engineering

Conclusions: Calibration Results

- Regression technique can be used to calibrate COCOMO locally using completed project data
- New cost drivers can be added and calibrated without destroying the structure of the COCOMO model
- COCOMO calibrated to local organization is more accurate then using generic COCOMO II model
- More project data is required to facilitate better calibration of generic COCOMO II model
- 1990's software data presents more challenges
  - Non-sequential processes: where are end-points?
  - Incremental development: how to separate the increments?
  - COTS, reuse, breakage, mixed language levels: what is size?

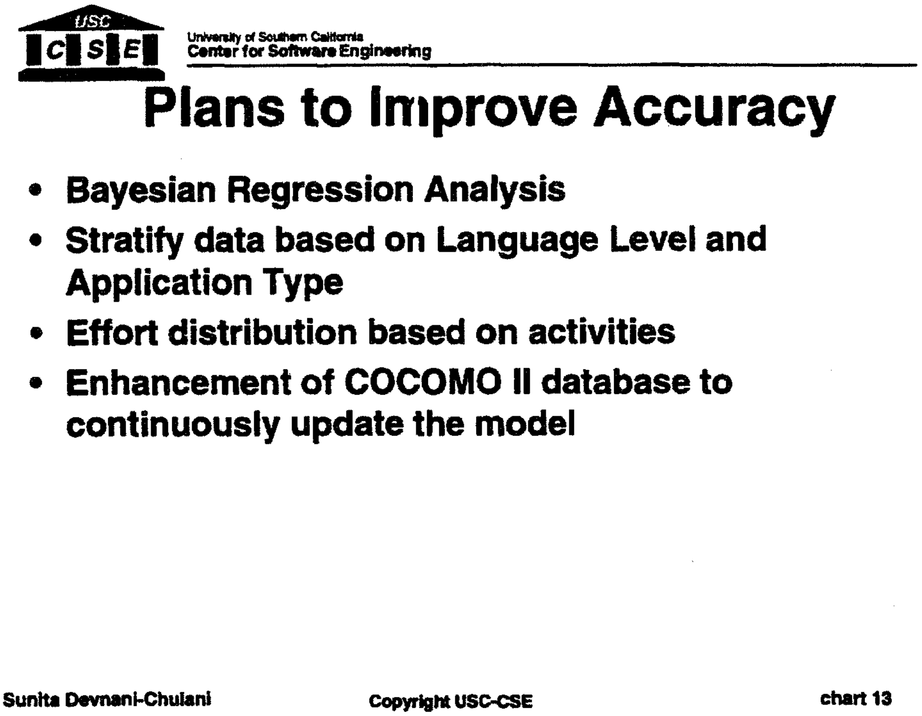
Sunita Devnani-Chulani chart 11

USC CSE University of Southern California
Center for Software Engineering

Presentation Outline

- COCOMO calibration
  Calibration process
  Results to date
- → Plans to Improve Accuracy
- Information Sources

Sunita Devnani-Chulani chart 12

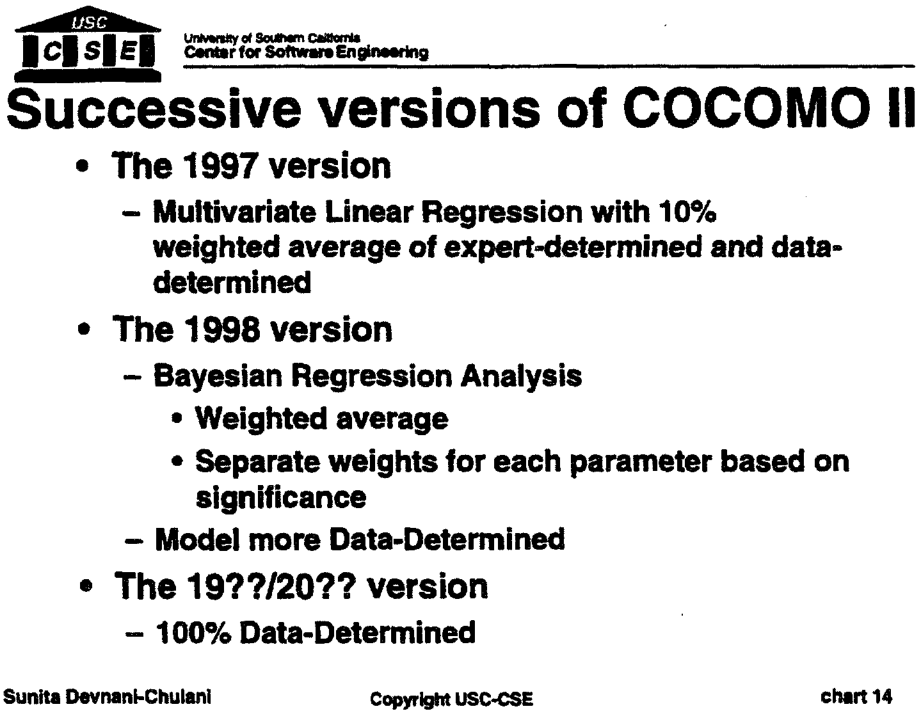


University of Southern California
Center for Software Engineering

Plans to Improve Accuracy

- Bayesian Regression Analysis
- Stratify data based on Language Level and Application Type
- Effort distribution based on activities
- Enhancement of COCOMO II database to continuously update the model

Sunita Devnani-Chulani   Copyright USC-CSE   chart 13



University of Southern California
Center for Software Engineering

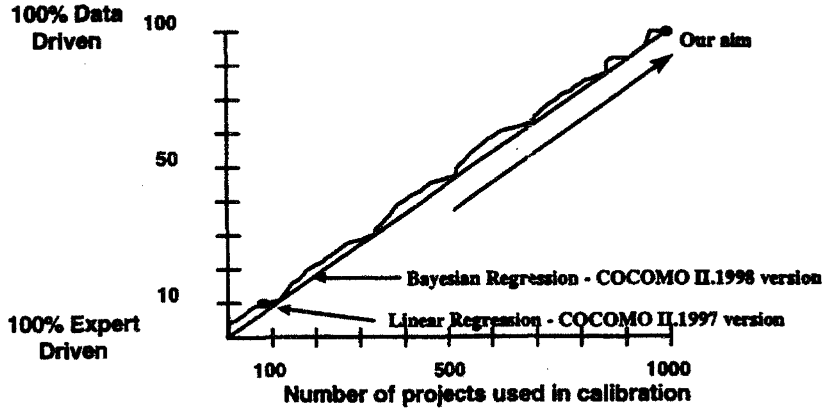
Successive versions of COCOMO II

- The 1997 version
  - Multivariate Linear Regression with 10% weighted average of expert-determined and data-determined
- The 1998 version
  - Bayesian Regression Analysis
    - Weighted average
    - Separate weights for each parameter based on significance
  - Model more Data-Determined
- The 19??/20?? version
  - 100% Data-Determined

Sunita Devnani-Chulani   Copyright USC-CSE   chart 14

University of Southern California
Center for Software Engineering

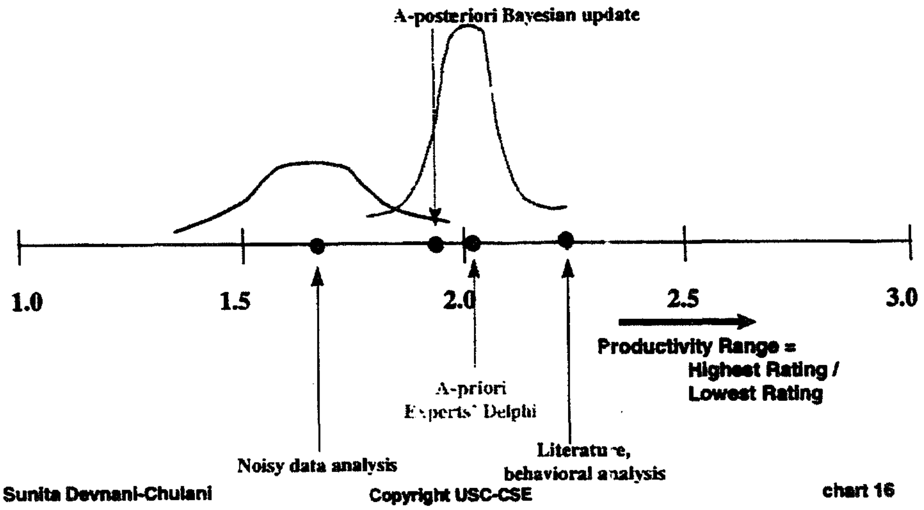
Evolving Model Values



Sunita Devnani-Chulani Copyright USC-CSE chart 15

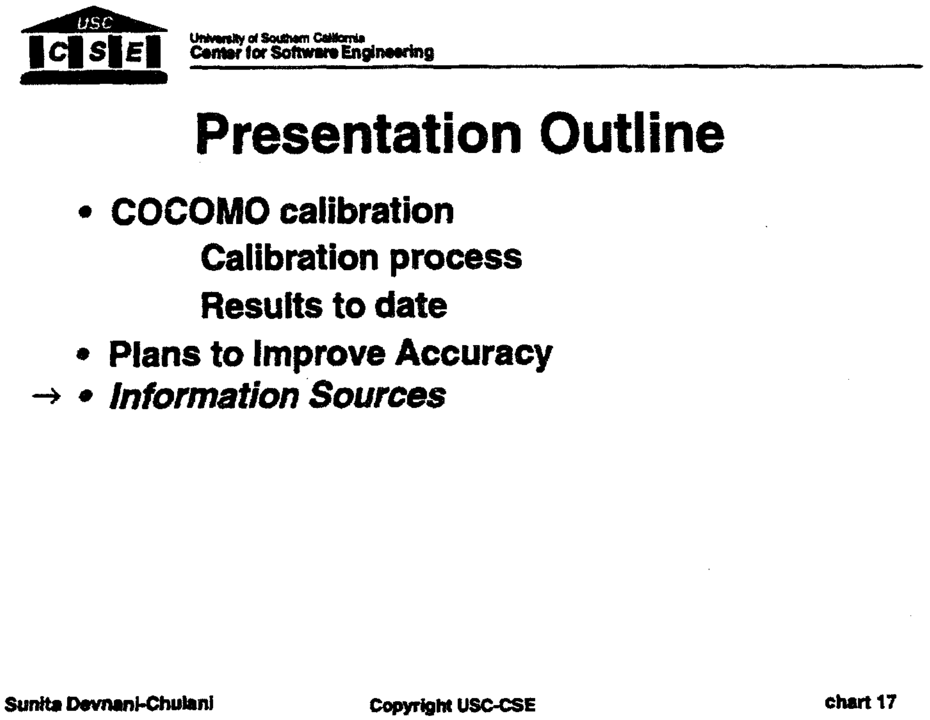
University of Southern California
Center for Software Engineering

Bayesian Approach



Productivity Range = Highest Rating / Lowest Rating

Sunita Devnani-Chulani Copyright USC-CSE chart 16

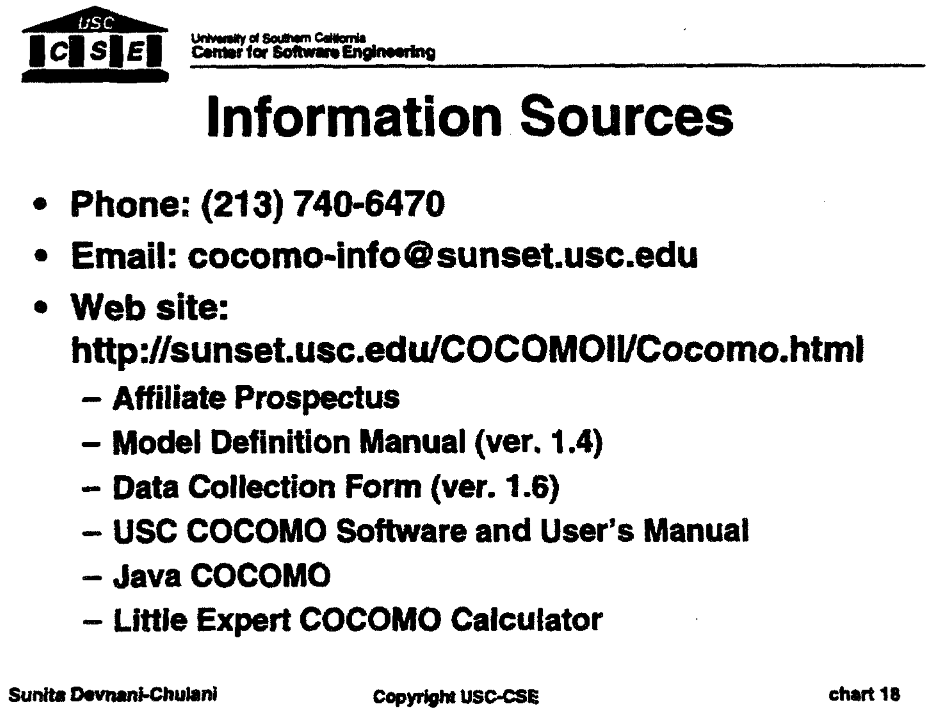


University of Southern California
Center for Software Engineering

Presentation Outline

- COCOMO calibration
  - Calibration process
  - Results to date
- Plans to Improve Accuracy
- → *Information Sources*

Sunita Devnani-Chulani Copyright USC-CSE chart 17



University of Southern California
Center for Software Engineering

Information Sources

- Phone: (213) 740-6470
- Email: cocomo-info@sunset.usc.edu
- Web site: http://sunset.usc.edu/COCOMOII/Cocomo.html
  - Affiliate Prospectus
  - Model Definition Manual (ver. 1.4)
  - Data Collection Form (ver. 1.6)
  - USC COCOMO Software and User's Manual
  - Java COCOMO
  - Little Expert COCOMO Calculator

Sunita Devnani-Chulani Copyright USC-CSE chart 18

Assessing the Accuracy of the COCOMO II.1997 Estimating Model

Donald J. Reifer, President
Reifer Consultants, Inc.
P. O. Box 4046
Torrance, CA 90510-4046
Phone/Fax: 310-530-4493
e-mail: d.reifer@ieee.org

52-61

360861

Abstract: The COCOMO II.1997 software cost model recently made it debut from the University of Southern California[1]. This new version of the popular estimating package popularized by Dr. Barry Boehm[2] has been redefined to incorporate many state-of-the-art features and capabilities aimed at making it usable well into the 21st century. While designed for upward compatibility, the model differs from its predecessor in both structure and mathematical formulation. For example, it uses five scale factors instead of three modes to compute the power law to which the base equation in the model is raised based upon function points or a software line of code size metric. As another example, it computes its effort multiplier from the model's base equation using 17 cost drivers, 3 of which are new to the model.

The goal of this presentation is to summarize a detailed analysis conducted to assess the estimating accuracy of the COCOMO II.1997 model. To accomplish this goal, we converted 91 existing estimates made with the previous version of the model using the Rosetta stone[3] that was developed for that purpose. The Rosetta stone provides its user with guidance on how to rate scale factors and cost drivers as estimates are converted from the COMOCO 81 to COCOMO II.1997 model or vice versa. The estimates using both models were then compared to each other and the actuals realized for 89 of the projects in the database. The other 2 data points were eliminated because they represented outliers that weren't significant. Results were then analyzed further using statistical methods to determine whether they were correlated or if there were co-linearity.

The presentation will assume that participants at the 22nd Annual Software Engineering Workshop know nothing about the COCOMO II-1997 model. It will start by briefly summarizing its features and how it differs from the original COCOMO. The presentation will then switch gears and discuss the accuracy analysis. It will highlight the approach taken and the results achieved. The presentation will conclude with a discussion of what the results mean. It will talk what features of the model seem to work based upon the data and where caution needs to be exercised because validity can not be determined. Finally, the presentation will make a plea for more data. The COCOMO team needs it to firm up their calibration and make the model which is publically available for free from the USC Center for Software Engineering web site more valid across a wider spectrum of applications domains.

----

[1] Barry W. Boehm, et. al., "The COCOMO 2.0 Estimating Model: A Status Report," ***American Programmer***, July 1996.

[2] Barry W. Boehm, ***Software Engineering Economics***, Prentice-Hall, 1981.

[3] Donald J. Reifer, ***Software Economics***, Reifer Consultants, 1997.

More Details for the Program Committee's Assessment

The COCOMO II.1997 Model

While the COCOMO II.1997 is of the same form as the original, it differs in both structure and content as follows:

	COCOMO 81	COCOMO II.1997
Structure	Single model aimed at waterfall paradigm	Three staged models supportive of incremental, spiral and other modern paradigms
Base Effort Equation	$\text{Effort} = a\,(\text{Size})^B \prod \text{EM}_i$	$\text{Effort} = a\,(\text{Size})^B \prod \text{EM}_i$
Power law (B)	Three modes: - Organic: B=1.05 - Semidetached: B=1.12 - Embedded: B=1.20	Five scale factors: $B = \sum sf_i$ and $1.01 \leq B \leq 1.25$ - Architecture/risk resolution
Size	Deliverable Source Instructions	Object Points, Function Points and Source Lines of Code
Effort Multiplier (EM)	15 cost drivers Platform: TIME, STOR, TURN, VIRT Product: RELY, DATA, CPLX Personnel: ACAP, AEXP, PCAP, LEXP, VEXP Project: TOOL, MODP, SCED	17 cost drivers Platform: TIME, STOR, PVOL Product: RELY, DATA, DOCU, CPLX, RUSE Personnel: ACAP, AEXP, PCAP, PEXP, LTEX, PCON Project: TOOL, SCED, SITE
Other Major Changes	Models for maintenance and adapted code	Updated models for maintenance, reused and adapted code

The Experiment

The author took data that he had from past estimates and actuals from 91 projects. Data were selected to be current (not over 7 years old), valid (the estimate and actuals data had been normalized and validated prior to entering it into the database) and representative of domains of interest to conference attendees (7 domains representative of flight and ground projects were chosen). The resulting database was then tested for homogeneity and co-linearity within and across domains. Outliers were identified and discarded because they biased the dataset. Each estimate was then converted to COCOMO II using a Rosetta stone developed for that purpose and run using the USC COCOMO II.1997.1 cost estimating package. The resulting COCOMO II and original estimates were then compared to the actuals realized for effort and the standard error was computed. Additional tests were run to look at correlation between independent variables used within the database and results were documented. It is increasing to note that only in 7 cases did the estimates exceed the actuals. Within this database, at least, this means that the models tend to estimate low.

The COCOMO Rosetta Stone

In order to convert from COCOMO 81 to COCOMO II.1997 and vice versa, a set of guidelines were created of the following form:

ADJUST SIZE AND SCOPE---->
ADJUST SCALE FACTOR---->
ADJUST EFFORT MULTIPLIER---->
COMPUTE EFFORT---->
ADJUST DURATION

The details of the conversion factors have been left out on purpose because we do not want to make them public until they are presented at the conference.

The Results

The results of the experiments conducted are summarized below in term so the accuracy of the effort estimate (ability to predict actuals) in different domains can be ascertained:

	No. of Data Points	**COCOMO 81 vs. Actuals (Percent Error)**	**COCOMO II vs. Actuals (Percent Error)**	**COCOMO 81 vs. COCOMO II (% Difference)**
Avionics	12	28	20	12
Business	18	18	11	15
Command & Control	13	31	17	22
Scientific	10	15	9	10
Simulators	9	25	14	18
Software products	14	9	9	8
Telecommunications	13	16	10	14

Of course, more details on the analysis conducted will be presented at the conference if time permits.

The Implications and Lessons Learned

The results of these experiments are significant. They demonstrate that COCOMO II has an improved accuracy when compared to the original estimating model when a standard process (the Rosetta Stone) is used to convert one set of model parameters to the other. They also prove that you can estimate software costs within ± 20% of actuals using the COCOMO model. Of course, the results are based upon a limited dataset and may not be indicative of what can happen in one organization or another. But, at least they offer managers hope that increased predictability and control over costs can be achieved in the foreseeable future.

Assessing the Accuracy of the COCOMO II.1997 Estimating Model

3 December 1997

Reifer Consultants, Inc.
P.O. Box 4046
Torrance, CA 90510-4046

Summary of the Effort

- This presentation summaries the results of an effort we undertook to estimate the accuracy of the COCOMO II.1997 cost model
- It concludes that you can achieve acceptable levels of accuracy by stratifying your data
  - Using domains, size and other globbings
- It demonstrates the usefulness of the USC model package auto-calibrate feature and the COCOMO Rosetta stone

A Little About COCOMO II.1997

- Develop software resource estimating model tuned to the life cycle practices of the 1990's
  - Key structure to future software marketplaces
- Develop a cost database and tool capability aimed at continuous model improvement
  - Provide quantitative framework for evaluating effects of software technology improvements
- Preserve the openness of original COCOMO and provide for upward compatibility

Major Model Changes

- Multi-model coverage
  - Three models replace one
- Flexibility in size inputs
  - SLOCs, function points or object points
- Requirements volatility replaced by % breakage
- Scaling factors replaces modes
- Cost driver changes
  - Several new factors and many updated values
- Revised schedule estimation model

Three Phase Model

- Application Composition Model
  - Involves prototyping to resolve risks
- Early Design Model
  - Exploration of alternative architectures
  - Uses function points and set of 7 cost drivers
- Post Architecture Model
  - Actual development & maintenance of software
  - Uses variety of size measures, 17 cost drivers and 5 scale factors to estimate resources

Post-Architecture Model Equations

Effort Equation

$$PM = \prod_{i=1}^{17} (EM)_i \, A\left[\left(1 + \frac{\underline{BRAK}}{100}\right) Size\right]^B$$

Scale Factor

Where:

$$B = 1.01 + \sum_{j=1}^{5} SF_j$$

Duration Equation

$$\text{Duration (months)} = [C\ (\text{Effort})^{(0.33+0.2(B-1.01))}]\left[\frac{\underline{SCED\%}}{100}\right]$$

COCOMO II.1997 Reuse Model

$$ESLOC = \frac{ASLOC\,[AA + AAF(1 + 0.02(SU)(UNFM))]}{100} \qquad AAF \leq 0.5$$

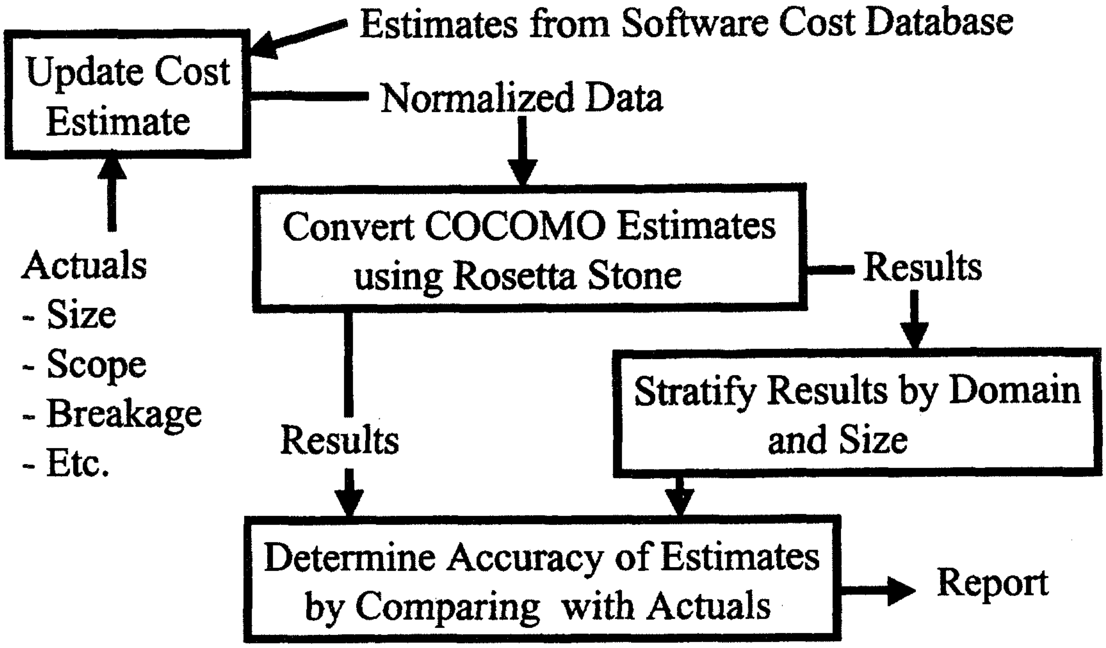
$$ESLOC = \frac{ASLOC\,[AA + AAF + (SU)(UNFM)]}{100} \qquad AAF > 0.5$$

Where: AAF = 0.4 (DM) + 0.3 (CM) + 0.3 (IM)
SU = Software Understanding
(zero when DM = 0 & CM = 0)
UNFM = Programmer Unfamiliarity
AA = Assessment and Assimilation
ASLOC = Adapted SLOC
ESLOC = Equivalent new SLOC

Advertised Predictive Accuracy

- COCOMO 81
  - Basic version (no cost drivers): within 30% of actuals, 29% of the time
  - Intermediate version: within 20% of actuals, 68% of the time
- COCOMO II.1997
  - Before stratification: within 25% of actuals, 47% of the time
  - After stratification: within 25% of actuals, 56% of the time

Our Calibration Approach



Estimates from Software Cost Database

Update Cost Estimate

Normalized Data

Actuals
- Size
- Scope
- Breakage
- Etc.

Convert COCOMO Estimates using Rosetta Stone

Results

Stratify Results by Domain and Size

Results

Determine Accuracy of Estimates by Comparing with Actuals

Report

Determining Initial Calibrations

- USC package has an auto-calibration feature
  - Permits you to compute effort and schedule constants based upon your cost histories
  - USC calibrates the cost drivers/scale factors
- To use these features, you must convert your COCOMO 81 files to COCOMO II.1997
  - Used Rosetta stone to do this and then adjusted results based upon knowledge of projects
- For new projects, must develop estimates

The Rosetta Stone

- Means to convert COCOMO 81 estimates to COCOMO II and vice versa
  - Provides mappings between model cost drivers
  - Provides mappings between modes/scale factors
- Many of the norms for the existing model cost drivers have changed (TOOL, etc.)
- Model also uses new cost drivers (RUSE, SITE, etc.) and new ways of addressing software engineering concepts (BRAK, MODP, etc.)

Example Rosetta Stone Settings

	Organic	Semi-Detached	Embedded
PREC	Extra High	High	Low
FLEX	Extra High	High	Low
RESL	Extra High	High	Low
TEAM	Extra High	Very High	Nominal
PMAT	Low	Low	Low
Rating	1.05	1.12	1.20

Results of Experimentation

- Using Rosetta stone (no adjustments, selected projects)
  - Effort: 4.11 versus 2.45 (scale factor at 1.01)
  - Schedule: 3.67 versus 2.66 (scale factor at 0.33)
  - Accuracy: within 25% of actuals, 49% of time
- Using Rosetta stone and knowledge base adjustments (all 89 projects in database)
  - Effort: 3.96 - Schedule: 3.43
  - Accuracy: within 25% of actuals, 56% of time

More Accuracy Analysis

- Using Rosetta stone, adjustments and domains

	Effort	**Schedule**
– Aerospace A	4.46	5.14
– Aerospace B	5.45	4.68
– Aerospace C	4.76	4.68
– Management Info Systems	2.65	2.76
– Software Packages	2.88	2.89
– Switching Systems	3.45	3.74

- Accuracy: improved to within 20% of actuals, 63% of time (across all domains)

More Analysis Results

- Using Rosetta stone, adjustments, domains and size stratification:

	Effort	Schedule
– Small (≤ 25KSLOC)	2.92	2.88
– Medium (≤ 150KSLOC)	3.14	3.33
– Large (> 150KSLOC)	4.22	2.66

- Accuracy: improved to within 20% of actuals, 68% of time (across all domains)

Final Experimental Results

	No. of Data Points	COCOMO 81 vs. Actuals (Percent Error)	COCOMO II vs. Actuals (Percent Error)	COCOMO 81 vs. COCOMO II (% Difference)
Avionics	12	28	20	12
Business	18	18	11	15
Command & Control	13	31	17	22
Scientific	10	15	9	10
Simulators	9	25	14	18
Software products	14	9	9	8
Telecommunications	13	16	10	14

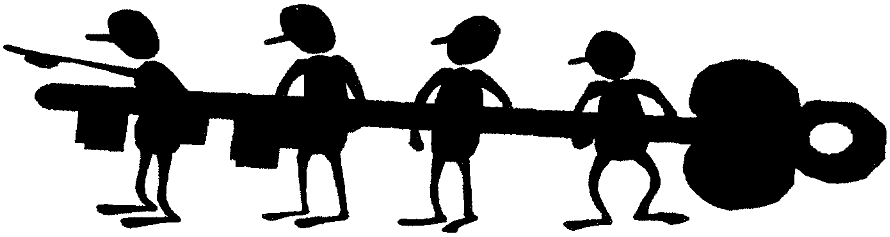
Using RCI database of 89 projects

Conclusions & Recommendations

- Accuracy improves when compared to original estimates when a standard process is used to convert one set of parameters to another
  - The Rosetta stone has value
- Statistical accuracy improves as a function of stratification by organization, domain, size, etc.
  - Reinforces importance of developing your own calibration
  - Package's auto-calibration feature is very useful

11/01/97

Final Thoughts



Models are getting better as more and more good data becomes available

11/01/97

360863

Component Based Software Development: Parameters Influencing Cost Estimation*

Randy K. Smith
Department of Computer Science
The University of Alabama
PO Box 870290
Tuscaloosa, Al 35487-0290
rsmith@cs.ua.edu
Voice (205) 348-3750
Fax (205) 348-0219

Allen Parrish
Department of Computer Science
The University of Alabama
PO Box 870290
Tuscaloosa, Al 35487-0290
parrish@cs.ua.edu
Voice (205) 348-6363
Fax (205) 348-0219

Joanne Hale
Area of MIS
Culverhouse College of Commerce and
Business Administration
The University of Alabama
Tuscaloosa, Al 35487
jhale@alston.cba.ua.edu
Voice (205) 348-0854
Fax (205) 348-0560

1 Introduction

Traditional software development is characterized by the structured programming paradigm introduced in the late 60's and early 70's. This paradigm relies on top-down functional decomposition to derive software modules (see Figure 1-a). The structured programming paradigm provides a monolithic view of the software development process [5]. Traditional software cost estimation models capture this monolithic view of software development. In these models, software cost is projected at the large-grained system level.

Contemporary development practices characterize a software application as interacting, independent components (see Figure 1-b). These components may be Commercial-Off-The-Shelf (COTS) components, internally developed reusable components or newly developed software artifacts. To accurately predict effort in Component Based Software Development (CBSD), a fine-grained approach is needed to identify and classify the relevant cost factors.

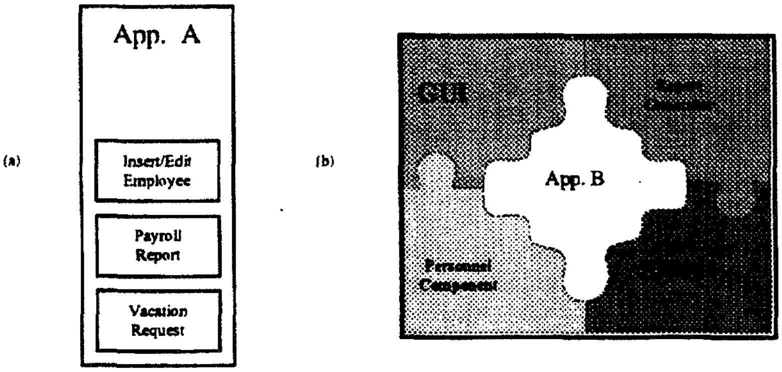


Figure 1 (a) Traditional Development, (b) CBSD

In recognition of the shift in development practice, monolithic cost estimation models have been adapted and refined to capture cost estimates for each component in an application [1, 2]. While such adaptations are useful, they do not capture all of the factors that could potentially impact component-based development [7, 8].

Section 2 demonstrates the application of a monolithic cost model (COCOMO) to CBSD. Section 3 identifies important metrics associated with CBSD. Section 4 discusses a preliminary study in which two of the CBSD metrics are used to augment the COCOMO model to produce a more accurate estimate for the project in the study. The conclusion section discusses the impact of these results and outlines additional work to be completed.

2 Using COCOMO for CBSD estimation

In his cornerstone book on software economics, Boehm details a comprehensive cost estimation model called COCOMO (Constructive Cost Model) [1]. The model provides three "levels" of application: basic, intermediate and advanced, based on the factors considered by the model. Each level of application is further divided into three modes of operation: organic, semi-detached and detached, based on the complexity of the application. The general effort model [1], applicable to all application levels and modes, is given by:

$$E = a(EDSI)^b X (EAF)$$

* Research supported in part by the National Science Foundation under grant #CDA-9712929

E is an effort estimation, expressed in man-months. EDSI refers to the numbet of Estimated Delivered Source Instructions. The parameters *a* and *b* are constants determined by the mode of development, both increasing with application complexity. The *effort adjustment factor*, EAF, is equal to 1 for the basic model and equals the product of 15 cost factors for the intermediate and advanced models. Each of the multiplicative cost factors is reflective of expected proportional increase (>1) or decrease (<1) in cost. For example, an application requiring a high degree of reliability may use 1.15 as one cost factor thereby increasing the overall effort estimate; an application that needs a low degree of reliability may use .88 as one cost factor, thereby decreasing the overall effort estimation.

Representing a monolithic approach to cost estimation, an example of using basic COCOMO to estimate the effort required in developing an 8500 line program in organic mode is given below.

$$E = 3.2\,(8.5)^{1.05} \text{ X } 1 = 30\,MM$$

Boehm also adopted the Intermediate COCOMO Model to apportion cost to individual components. Consider the 8500 line project again, made up of the components listed in Table 1.

Component	EDSI	% of total	CMM_{NOM}
Personnel	2000	23.4%	7.06
Invoice	3000	35.3%	10.60
Receivable	3500	41.2%	12.36

Table 1. Component Level Intermediate COCOMO

Based on the 30 MM computed above for E, the expected number of EDSI per man-month is given by:

$$(EDSI/MM)_{NOM} = 8500/30 = 283\ EDSI/MM$$

Using the EDSI/MM, each component is then apportioned its contribution to the total. For example, the nominal component man-month (CMM_{NOM}) for the personnel component is given by:

$$CMM_{NOM} = EDSI_{per\ Component} / (EDSI/MM)_{NOM} = 2000/283 = 7.06\ CMM_{NOM}$$

After calculating the CMM_{NOM} for each component, the *Effort Adjustment Factor* (EAF) is calculated individually for each component. The EAF factor is applied to CMM_{NOM}, arriving at a new adjusted MM estimate (CMM_{ADJ}) for each component. This is unlike the monolithic model, which applies a single EAF to the system as a whole. Thus, we are able to account for variance among the cost factors for the various components. For example, the CMM_{ADJ} for the invoice component is calculated by:

$$CMM_{ADJ} = (CMM_{NOM})(EAF) = (10.60)(1.13) = 11.98\ CMM_{ADJ}$$

Table 2 contains nominal effort estimations, EAFs and adjusted effort estimates for each component in our sample system. Note that by factoring the components individually through adjusted component man-months in Table 2, a new estimate of 31.24 man-months is found.

Component	EDSI	% of total	CMM_{NOM}	EAF	CMM_{ADJ}
Personnel	2000	23.4%	7.06	.89	6.28
Invoice	3000	35.3%	10.60	1.13	11.98
Receivable	3500	41.2%	12.36	1.05	12.98

Table 2. Component Level Intermediate COCOMO with Adjustments

3 A Model of CBSD

The primary characteristic of CBSD that is not captured by traditional cost estimation models is the effect of scheduling at the component level. Components may be developed early or late relative to a given project; also the schedule for a given component may be compressed or relatively spread out. In our work, we treat time as a "snapshot" in order to examine the state of the system. A snapshot consists of one "time-unit"; in our work, we consider the impact of varying the time intervals associated with one unit. In CBSD, a programmer can be working on multiple components at any given time. Conversely, at any given time a component may have multiple developers. During a single time unit, the project can have multiple components under development by multiple programmers. The following figure captures the project state at a single time unit (see Figure 2).

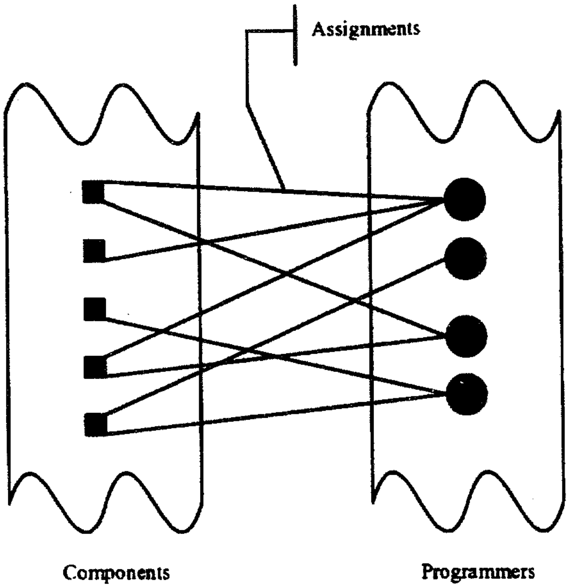


Figure 2. CBSD Activity at one time unit

We are assuming that time data are reported in terms of "activities." Each activity contains a *programmer id*, *component id* and *time unit*. Thus, we are able to determine when the various programmers worked on different components. Using this concept of components, the following terms can be defined. Each of these terms should be considered in the context of a specific component C.

- *Total Time Units*-The set of time units for which activity for C was reported.
- *Active Programmers*-The set of programmers reporting activity for C.
- *Active Time Units*-The subset of total time units where C has one or more active programmers (i.e., those time units in which some effort was expended on C).
- *Programmer Experience Set*-The set of components completed by C's active programmers prior to work beginning on C.
- *Project Experience Set*-The set of components completed by anyone prior to work beginning on C.

This expanded definition of CBSD captures the concurrent and varying nature of the environment. It allows the model to capture issues relating to time that are absent from monolithic cost models. This expanded model of CBSD allows for the exploration of fine-grained metrics to aid in predicting the cost in component-based software development.

Using the model and definitions given above, a set of metrics can be defined to capture parameters in CBSD that are not present in the Intermediate COCOMO model. In this section, a suite of metrics is proposed that are a direct development of the CBSD model presented above. The metrics are given in the table below.

Intensity	The ratio of the quantity of active time units to quantity of total time units for a particular component.
Concurrency	Number of programmers working simultaneously on a component, averaged over all active time units.
Fragmentation	The degree to which a programmer's time is fragmented over multiple components. Specifically, for all active programmers of a given component C, it is the number of different components for which those programmers reported activity, averaged over all active time units for C.
Project Experience	The cardinality of the project experience set for a particular component.
Programmer Project Expereince	The cardinality of the programmer experience set for a specific component with respect to all active programmers of the component.
Team Size	Total number of active programmers for a specific component.

Table 3. Proposed CBSD Metrics

Using this extended model of CBSD, an empirical study was undertaken to examine some of the metrics highlighted above. The study seeks to develop a more accurate model of cost estimation in CBSD. Investigation of the remaining metrics is currently in progress.

4 Empirical Study

A component-based software cost estimation model is being developed from data provided by a regional firm that specializes in information management solutions for state governments. The firm is currently under contract to provide the state government with a client-server solution to replace a COBOL-based ISAM database system. The legacy system contains over 3000 screens and 1 million lines of COBOL code. The new solution will incorporate client-server databases and end-user systems deploying a contemporary windowing operating systems. The new system will support in excess of 400 users distributed through a statewide network. The system will support departmental accounting, a large-scale bidding system for state infrastructure, inventory control and departmental personnel systems.

The 16-member development team has duties distributed among software development, computer networking and database administration. The development environment consists of a series of networked workstations utilizing a windowing operating system. The application development environment consists of COTS libraries, application generators, database systems, and internally developed domain specific reuse libraries.

4.2 Methodology

There are two types of data examined in this study: product data and process data. The product data for this study were obtained from Hierarchical-Input-Process-Output (HIPO) diagrams for each of over 400 components. Using the preliminary HIPO diagrams, Unadjusted Function Points (UFP) were computed for each HIPO using the counting rules outlined by Dreger [3]. Function Points are a technology independent measure of the functionality of a system as seen by the user [4, 6].

The process data are obtained from the time accounting system used for billing. The data includes a daily activity log giving:

- The date the work was done.
- The task number on which the work was done.
- The identification number of the employee performing the work.
- The number of hours worked in quarter hour increments.

From this log, a mapping is made from task number to HIPO number and from the employee identification number to employee job title. The job title is used as a relevant estimate of experience.

Using this data, we identified a subset of the metrics given above to be used in this study. The metrics used are:

- Programmer project experience (PEXP).
- Team size (TEAM).

These parameters coupled with the UFP estimates gave us a preliminary cost estimation model for component-based software development.

In order to compare the predictive ability of the component-based parameters against a known model, organic-mode Intermediate COCOMO estimates were calculated for the components of this project. An additional measure of programmer experience was obtained by surveying the team in regards to their experience in general and their experience with the various factors involved in this development environment. The experience data is used by the Intermediate COCOMO model. The Unadjusted Function Point was used to determine Lines of Code estimates for input into the Intermediate COCOMO model. The estimation for Lines of Code from Function Points used the C++ multiplication given by Jones [6]. After determining a Lines of Code estimate, each HIPO was examined to determine the impact of the 15 cost factors required by the Intermediate COCOMO Model. Effort estimations were made for each component using the Component Level Estimation Form (CLEF) given by Boehm [1].

Using the process data collected from the billing system, we analyzed the identified parameters that pertain to component-based software development, developing an augmented COCOMO model using the PEXP and TEAM parameters. This augmented model gave us better predictions than the standard Intermediate COCOMO model. The results of this augmented model are discussed in section 4.3.

4.3 Results

A set of 89 components was utilized to perform this preliminary analysis. The proposed model augments the Intermediate COCOMO model with the parameters relevant to component-based software development. In order to obtain a base-line measurement, a regression analysis was performed comparing the predictive ability of COCOMO to the actual hours worked. Table 5 gives the R^2 and standard error for this initial analysis. Using the COCOMO estimates, the tested model is given by:

$$HOURS = \alpha + \beta_1(COCOMO) + \beta_2(PEXP) + \beta_3(TEAM)$$

In addition to PEXP and TEAM which were defined above, HOURS refers to the number of hours devoted to a component, while COCOMO refers to the results of the Intermediate COCOMO model. The analysis of variance for the tested model is given in Table 4.

Source	Sum-of-Squares	df	Mean-Square	F-ratio	P
Regression	234432.171	3	78144.057	26.024	<0.001
Residuals	255238.520	85	3002.806		

Table 4. Analysis of Variance

The regression yielded the following equation:

$$HOURS = -39.202 +7.641(COCOMO) +0.027(PEXP) + 23.981(TEAM)$$

The adjusted R^2 and standard error for the augmented model is given in Table 5.

	Regression Analysis Adjusted R^2	Standard Error of Estimate
Intermediate COCOMO	.394	58.063
Augmented Intermediate COCOMO (COCOMO + PREV + PROG)	.460	54.798

Table 5. Regression Analysis

As can be seen in Table 5, the augmented COCOMO model produced a better overall fit than the stand-alone COCOMO model. The addition of the two component-based parameters marginally decreased the size of the standard error of the estimates.

Another mechanism to judge the accuracy of a regression model is a measure of prediction quality. Fenton and Pfleeger summarize the prediction quality measure and give a rule of thumb that a good model should show that 75% of its predictions fall within 25% of the actual values [4]. Using the prediction quality measure, the augmented COCOMO model performed better than the stand-alone model, yet yielded a prediction quality measure of only 31% of the estimates falling within the 25% of the actual values.

For the data in this study, a new model that augments Intermediate COCOMO with the TEAM and PEXP parameters outperformed the standard COCOMO model. These two additional parameters are easily captured early in the software process and are relevant to component-based software development.

5 Conclusion

Cost estimation for modern software development practices can benefit from a fine-grained approach. The traditional monolithic approach to software cost estimation can hide and mask both positive and negative cost factors in Component-Based Software Development. The Intermediate COCOMO model has been offered as a model for predicting development effort for software components. In this study, the best performance was achieved using the Intermediate COCOMO model augmented with the number of programmers assigned to a component and the number of components previously developed. Both models examined in this study performed poorly on the prediction quality criterion. Both models had high relative errors. Thus, more research is needed to develop cost estimation models that are suitable for the rapidly-growing component-based software development environment.

The parameters examined in this study mark an initial attempt at identifying the cost factors uniquely associated with component-based software development. These parameters were initially selected because they possess characteristics that are attractive in a cost model:

- They are quantitative values.
- The parameters fall along a real value scale.
- They are easily derived during the design stage.
- They characterize component-based development and are not fully captured by traditional monolithic cost models.

For this study, the Function Points were derived from HIPO diagrams, a work product from the design stage. The regressions used to identify and examine these parameters are performed on data routinely collected during the software development process. The employee information is obtainable from work histories. The development ordering is readily available from a critical-path or PERT analysis.

Monolithic cost models are designed to estimate monolithic software. The modern practice of taking advantage of COTS, application generators and extensive reuse requires a finer-grained approach than is available in traditional models. The parameters outlined in this study are only indicative of the types of cost factors needed. The parameters from this study performed well for the application environment used during this study. Each organization must examine its software development practices and identify the relevant parameters in their organization.

References

1. Boehm, B., *Software Engineering Economics*, Prentice-Hall, Inc., 1981.

2. Boehm, B., Clark, B., Horowitz, E., "Cost models for future life cycle processes: COCOMO 2.0", *Annals of Software Engineering*, vol. 1, no. 1, November, 1995, pp. 1-24.

3. Dreger, J., *Function Point Analysis*, Prentice-Hall, Inc., 1989.

4. Gries, D., "On Structured Programming" in *Software Design Strategies*, Bergland, G., Gordon, R., eds., IEEE Computer Society, 1981.

5. Fenton, N., Pfleeger, S., *Software Metrics: A Rigorous and Practical Approach. Second Edition*, PWS Publishing Company, 1997.

6. Jones, C., "Backfiring: Converting Lines of Code to Function Points", *Computer*, vol. 28, no. 11, November, 1995, pp. 87-88.

7. Leonhard, C., Davis, J., "Job-Shop Development Model: A Case Study", IEEE Software, vol. 12, no. 2, March, 1995, pp.86-92.

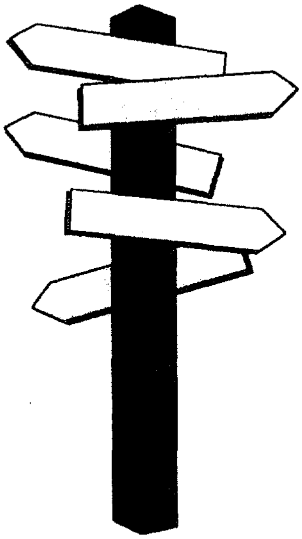
8. McConnell, S., "Less is More", *Software Development*, vol. 5, no. 10, October, 1997, pp. 28-34.

Component Based Software Development:
Parameters Influencing Cost Estimation

Randy Smith, Allen Parrish, Joanne Hale
The University of Alabama

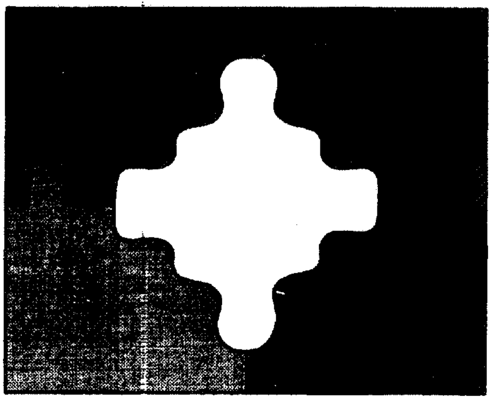
Introduction

- Traditional Development
- Component Based Software Development
- A Model for CBSD
- Research Study
- Results
- Conclusion



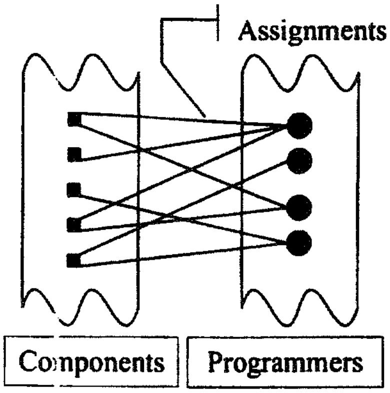
Component Based Software Development

- Software by Composition
- Reuse
  - All aspects of development
- COTS, MOTS GOTS
- Standards
- Producers & Consumers



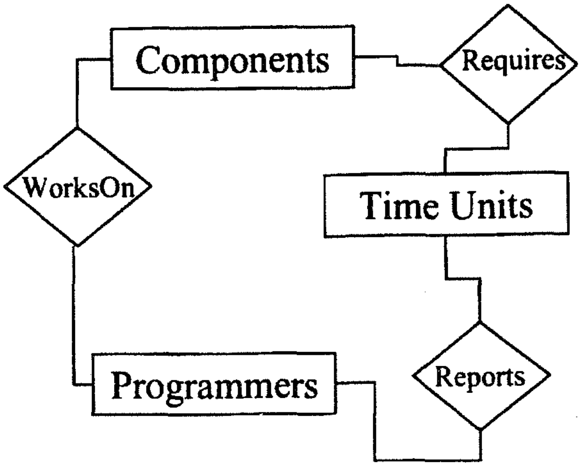
A Model of CBSD

- Producer POV
  - Multiple Components
  - Multiple Programmers
  - Time
    - Dynamic
    - Missing from traditional effort models



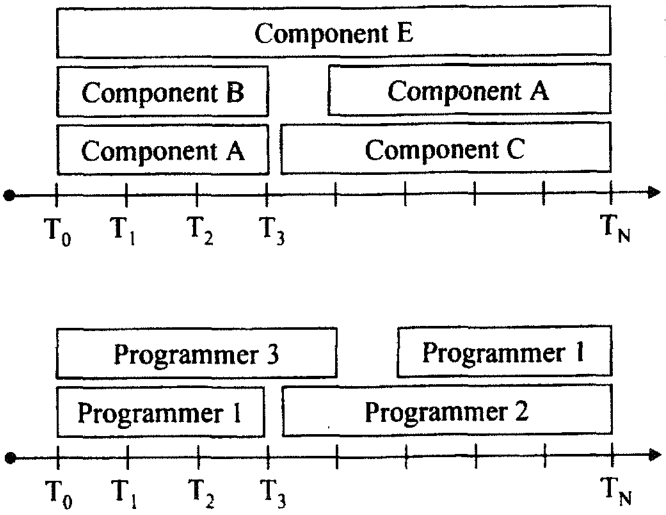
A Model of CBSD

- Components
  - Set C
    - Attributes
      - Reuse
      - Size
- Programmers
  - Set P
    - Attributes
      - Experience
- Time
  - Set T



A Model of CBSD

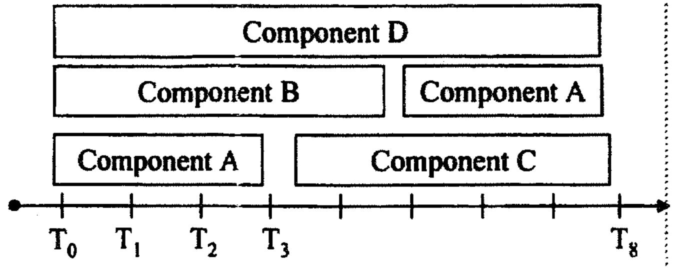
- **Definitions**
  - **Active Components**
    At T_2 = {A,B,E}
  - **Component Time Span**
    For Component A ={T_0. T_N}
  - **Component Active Time Units**
    For Component A=7
  - **Experience Set**
    For Component C ={B}
  - **Active Programmers**
    At T_2 = {P1, P3}



A Model of CBSD

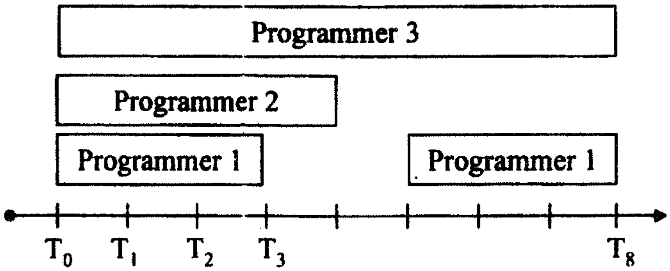
- *Intensity* -The ratio of the quantity of component active time units to component time span for a particular component.

  For Component A = 6/8



- *Concurrency* -Number of programmers working simultaneously on a component averaged over all active time units

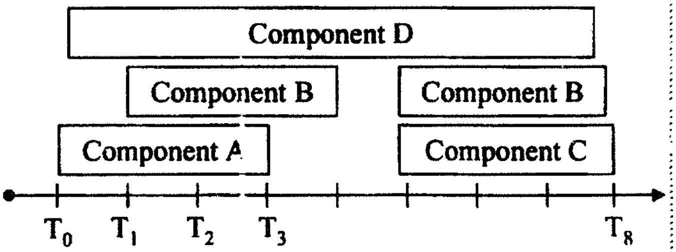
  For Component D = 22/9



A Model of CBSD

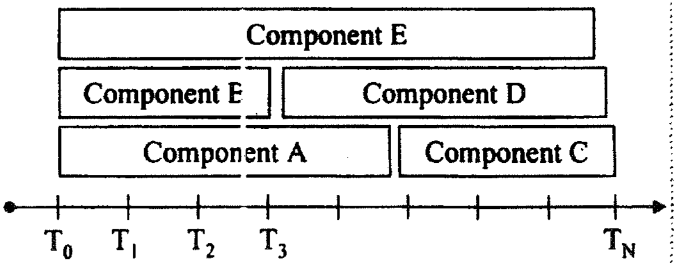
- *Component Project Experience* -The cardinality of the Experience Set for a particular component.

  For Component C = 1



- *Programmer Project Experience* -The cardinality of the Experience Set for a particular component with respect to an individual programmer.

  For Component C = 2



A Model of CBSD

- *Programmer Fragmentation* -The degree to which a programmers time is fragmented over multiple components.

	Work Reported	Work Reported	Work Reported
Programmer 1	C1, C2, C3, C4	C1, C2	C1
Programmer 2	C1, C2	C2, C4	C1
	T_1	T_2	T_3

Fragmentation at $T_1 = (4+2)/2 = 6/2 =3$
$T_2= (2+2)/2 = 4/2 = 2$
$T_3=(1+1)/2 = 2/2 = 1$

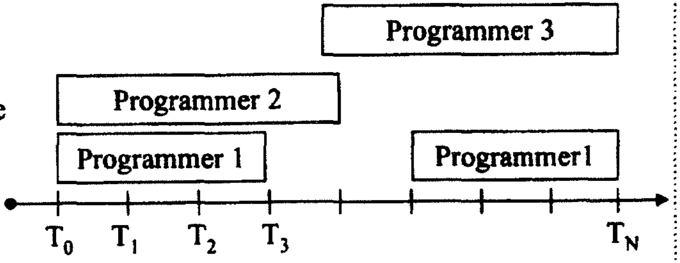
Fragmentation for C1 is 6/3 =2

9

A Model of CBSD

- *Team Size* -Total number of active participants over all time units for a particular component.

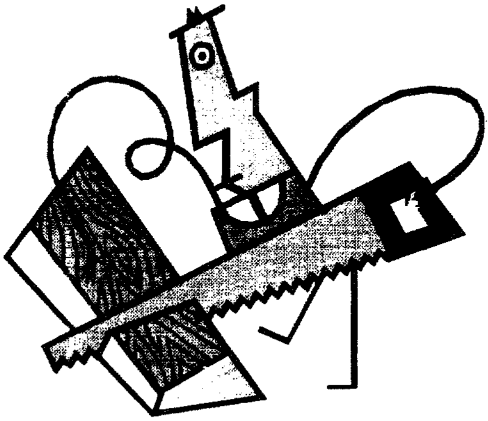
3 Programmers



10

Research Study

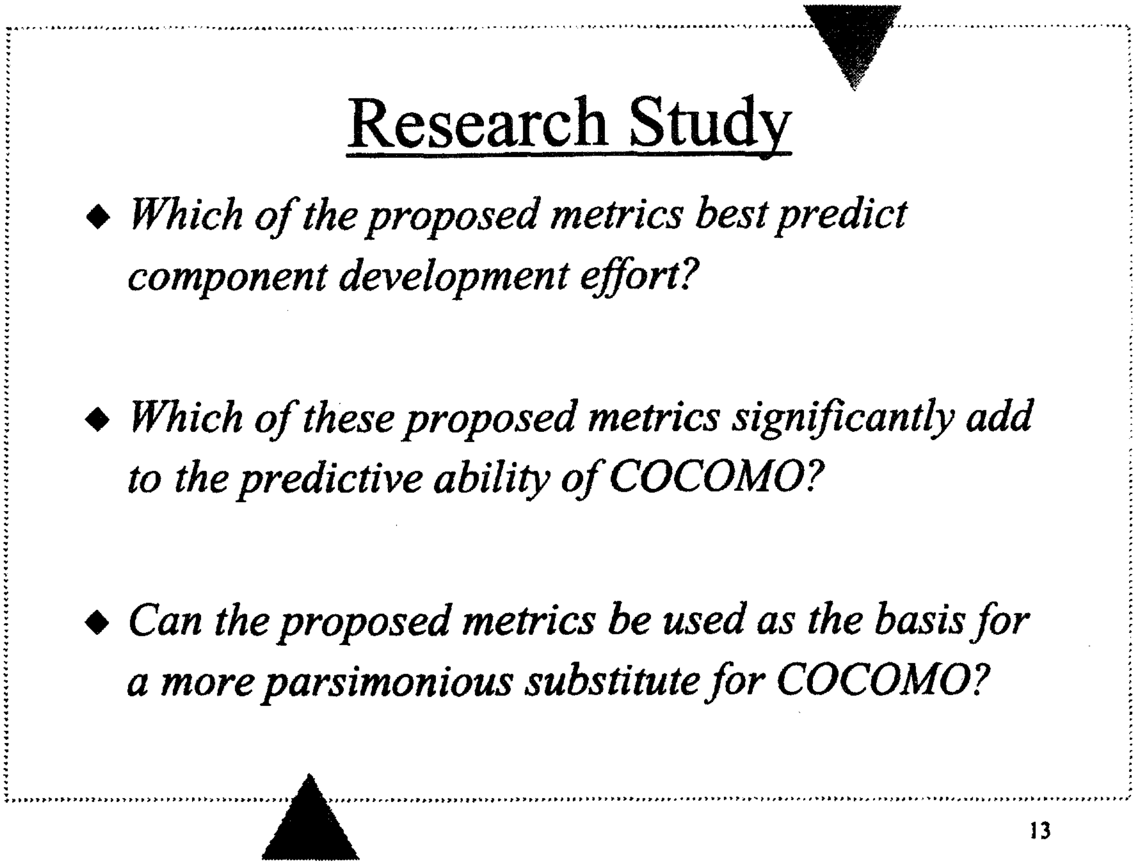
- Regional Contractor working for state government
  - Replace mainframe hosted legacy system
    - ▾ 1MLOC COBOL, 3K Screens
  - Client/Server, distributed
    - ▾ 400 users, state wide
  - Data Sources
    - ▾ HIPO diagrams
    - ▾ Time Accounting System



Research Study

- Product Data
  - Specified by Hierarchical-Input-Output diagrams (HIPOS)
  - 300 to 400 HIPOS
  - Size
    - ▾ Unadjusted Function Points
  - Reuse
    - ▾ Percentage of source line reused
  - Survey
    - ▾ Gross Experience
    - ▾ Environment Experience
  - Intermediate COCOMO

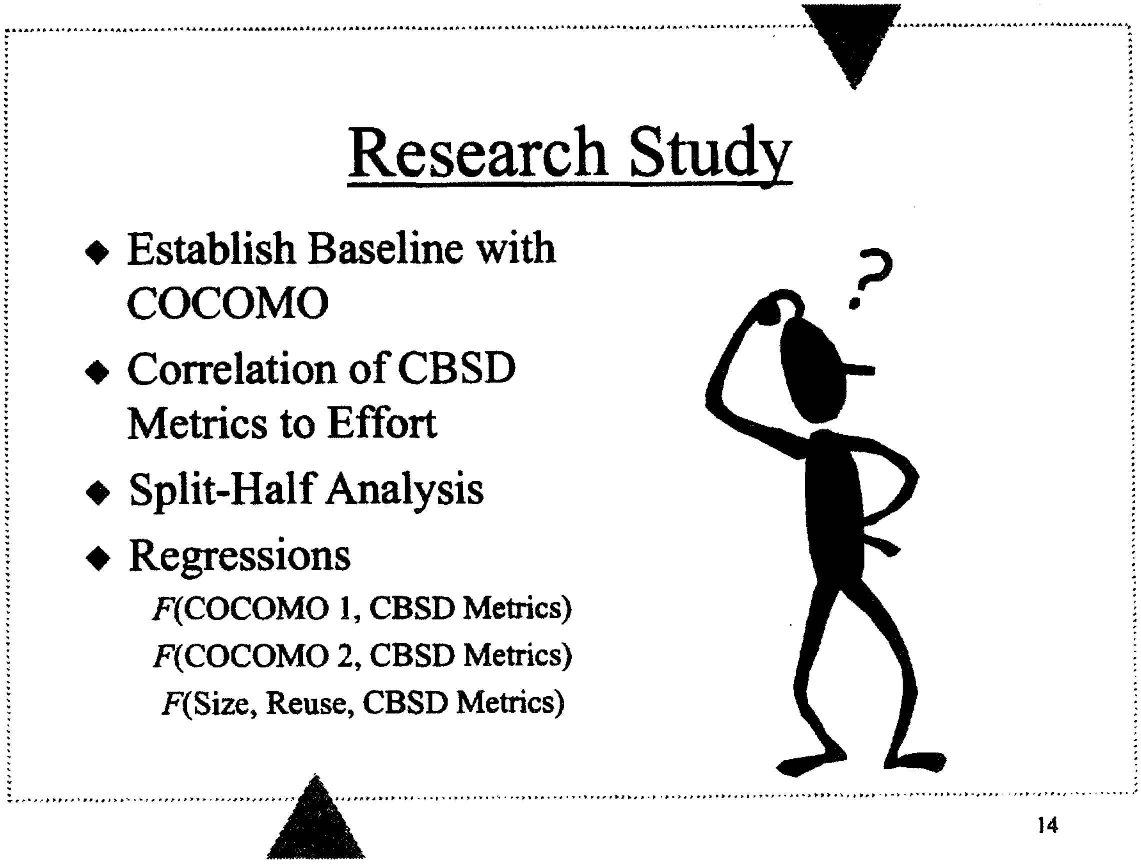
- Process Data
  - Intensity *(INTS)*
  - Concurrency *(CONC)*
  - Component Project Experience *(PEXP)*
  - Programmer Project Experience *(PRPE)*
  - Programmer Fragmentation *(PFRG)*
  - Team Size *(TEAM)*



Research Study

- *Which of the proposed metrics best predict component development effort?*
- *Which of these proposed metrics significantly add to the predictive ability of COCOMO?*
- *Can the proposed metrics be used as the basis for a more parsimonious substitute for COCOMO?*

13



Research Study

- Establish Baseline with COCOMO
- Correlation of CBSD Metrics to Effort
- Split-Half Analysis
- Regressions

  F(COCOMO 1, CBSD Metrics)

  F(COCOMO 2, CBSD Metrics)

  F(Size, Reuse, CBSD Metrics)

14

Preliminary Study

- 89 Components
- Intermediate COCOMO
- Augment Intermediate COCOMO
  - Two CBSD metrics

$$HOURS = \alpha + \beta_1(COCOMO) + \beta_2(PEXP) + \beta_3(TEAM)$$

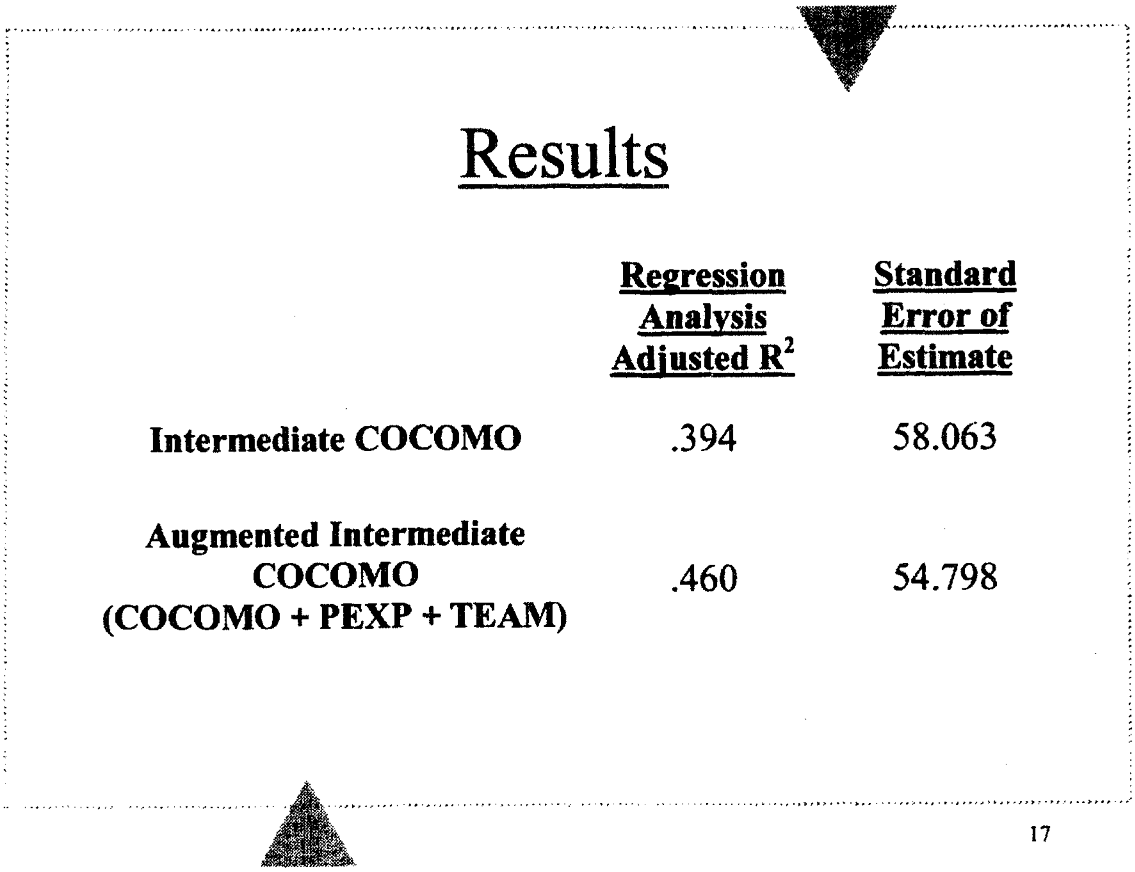
15

Results

Source	Sum-of-Squares	df	Mean-Square	F-ratio	P
Regression	234432.171	3	78144.057	26.024	<0.001
Included COCOMO PEXP TEAM					
Residuals	255238.520	85	3002.806		

The regression yielded the following linear model:
HOURS = -39.2 +7.6(COCOMO) +0.03(I'EXP) + 24(TEAM)

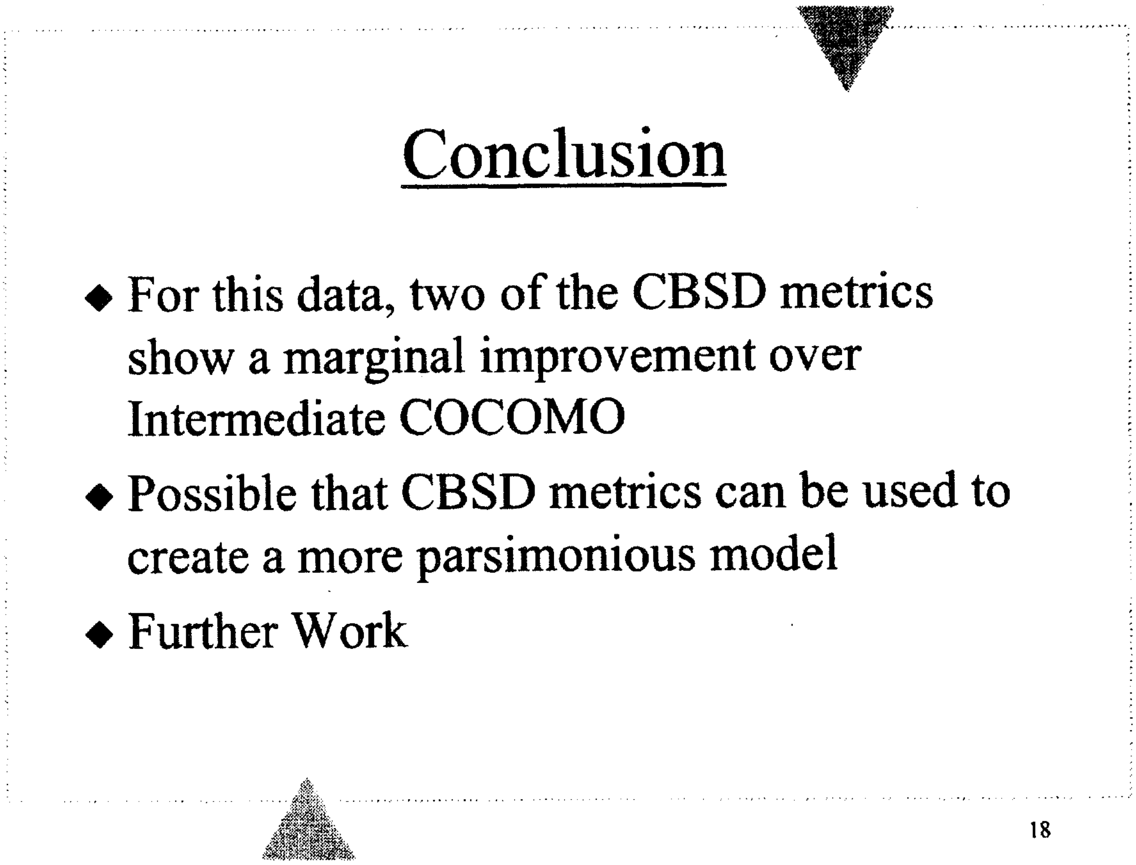
16



Results

	Regression Analysis Adjusted R^2	Standard Error of Estimate
Intermediate COCOMO	.394	58.063
Augmented Intermediate COCOMO (COCOMO + PEXP + TEAM)	.460	54.798

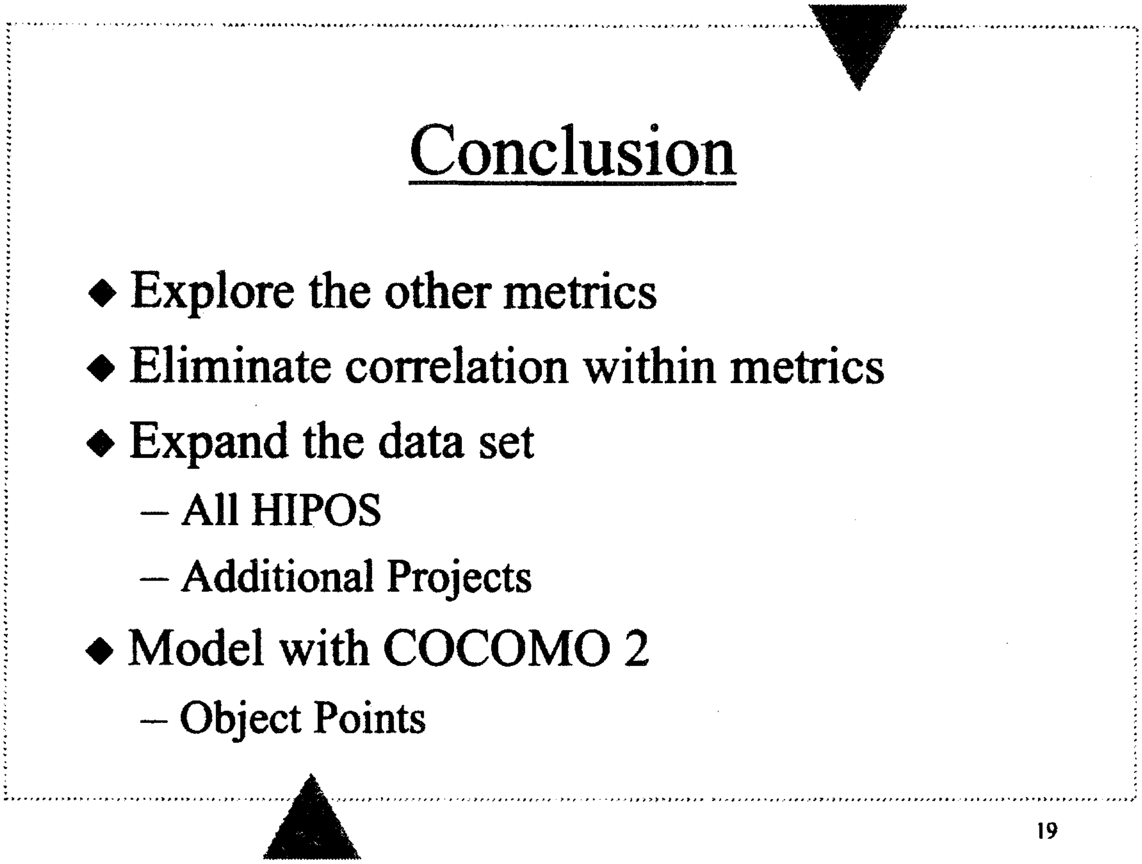
17



Conclusion

- ◆ For this data, two of the CBSD metrics show a marginal improvement over Intermediate COCOMO
- ◆ Possible that CBSD metrics can be used to create a more parsimonious model
- ◆ Further Work

18



Conclusion

- Explore the other metrics
- Eliminate correlation within metrics
- Expand the data set
  - All HIPOS
  - Additional Projects
- Model with COCOMO 2
  - Object Points

19

Page intentionally left blank

Page intentionally left blank

/

360766

A Verifier for Object-Oriented Designs[‡]

K. Periyasamy and W. Baluta
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
Ph: (204) 474-8682
Fax: (204) 269-9178
Email: {kasi, baluta}@cs.umanitoba.ca

[‡] This work was partially funded by a grant from Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Many object-oriented design methods such as Rumbaugh's [14], Booch's [2] and Coad and Yourdon's [6] methods use graphical notations which enable a software designer to describe a design more precisely and in an easily comprehensible manner. Tools supporting such object-oriented design methods generally provide a graphical user interface (GUI). The GUI will display a list of symbols corresponding to object-oriented design primitives and a working area. A designer using such a tool will develop object-oriented designs by selecting appropriate symbols from those provided by the GUI and by connecting them in a meaningful way.

One of the problems with existing tools for object-oriented design methods is that they merely act as drawing tools with minimal syntactic checking. For example, OMTool [10] which implements Rumbaugh's method, ensures that no two classes in a single class diagram have the same name. However, it is possible to develop a design diagram using OMTool in which two classes, A and B, inherit each other (i.e., A inherits B and B inherits A). Such cyclic inheritance is semantically invalid according to principles of object-orientation. A possible remedy for these problems is to embed verification strategies in the tool itself. This process requires the development of formal semantics of the modeling notations used by the tool.

In this paper, we describe an axiomatic semantics for the modeling notations used by the Object Modeling Technique (OMT) proposed by Rumbaugh and others [14]. We also describe the features of a verifier which implements the semantics. The modeling notations for which we have developed the formal semantics include class structure, its components (such as attributes and operations), relationships among classes (such as association and aggregation), object states, transitions between states and events. Most of these notations are also available in the Unified Modeling Language (UML) [3, 4], which is emerging as the industry-standard for object-oriented analysis and design. We have used the Z formal notation [15] to develop the semantics; the specification has been type-checked using the *fuzz* type checker [16]. The implementation of the semantics was carried out using a rule-based approach; we used the tool CLIPS [5], an industrially approved tool for developing knowledge-base systems for this purpose. The axioms in Z have been implemented as rules in CLIPS. All the rules have been constructed using templates with slots for generic parameters. When verifying a particular design, the design elements are substituted for the parameters of appropriate rules and the rules are fired.

Currently, we use a temporary format for input to the verifier. It is one of our future work to integrate the verifier with Rational Rose tool [13] so that the verifier accepts input of the same format as that of Rational Rose. Current version of Rational Rose (Rose 4.0) is intended for developing object-oriented designs using UML.

The rest of the paper is organized as follows: Section 2 describes the axiomatic semantics of OMT in Z. This includes the specification of object model and dynamic model. We describe the implementation of the semantics in Section 3. The paper concludes in Section 4 with comments on continuing work.

2 Axiomatic Semantics for OMT

The Object Modeling Technique (OMT) consists of three models: *object model*, *dynamic model* and *functional model*. Object model describes classes and objects, and static relationships among them such as aggregation and generalization. The dynamic model consists of state transition diagrams (also referred to as state diagrams for short in the rest of this paper), one for each class in the object model. Each state diagram describes the dynamic behavior of objects instantiated from the class for which the state diagram was developed. The functional model describes a functional view (as opposed to an object-oriented view) of the chosen application and typically uses data flow diagramming notations. In our work, we have discarded the semantics for functional model because it is not object-oriented. Moreover, UML does not include functional model.

Figure 1 shows the object model for a simple library management system. In OMT, a class is pictorially represented by a rectangle with one to three partitions; these partitions, from top to bottom, contains the name of the class, its attributes and its methods. The library system consists of two types of users, represented by the classes *Student User* and *Faculty User*. The class *User* is a generalization of *Student User* and *Faculty User*. The library contains three categories of items that can be borrowed: *Book*, *Journal* and *Collections*; the last category includes items different from books and journals such as CD-ROMS, video cassettes and class notes. The class *Item* is the superclass of the three classes *Book*, *Journal* and *Collections*. The two unnamed classes represent the associations *Hold* and *Bottom*; these two classes contain the link attributes [14, pages 31–32] of the respective associations.

Figure 2 shows the state diagram for the class *Book* in the library system. A round-edged rectangle in OMT pictorially indicates a state. Nesting of states is indicated by enclosing the rectangles corresponding to the nested states. In Figure 2, the state *InCirculation* encloses the states *OnShelves* and *LoanedOut*; *OnShelves*, in turn, encloses *NoHolds* and *Holds*. An arrow in a state diagram represents a transition. A transition is usually named (e.g., *loan out*), may contain parameters (enclosed in parentheses after the name of the transition) and may be associated with a condition (enclosed in square brackets after the name and parameters of the transition). An action associated with a transition is specified after the condition. This is indicated in the diagram with a symbol '/' followed by the action's name and its parameters; see the transition *loan out* in the state *LoanedOut* which leads to the same state.

2.1 Modeling Notations

In this section, we describe the modeling notations of OMT for which we have developed the semantics. For each notation, we give an informal description followed by its formal semantics. The informal descriptions have been adopted from UML. Due to space limitations, we give the descriptions and semantics for only a subset of the notations; readers are referred to [1] for an exhaustive list. In the specification, we use several types, relations, functions and global declarations. All of them have been specified in [1], but we include only a subset

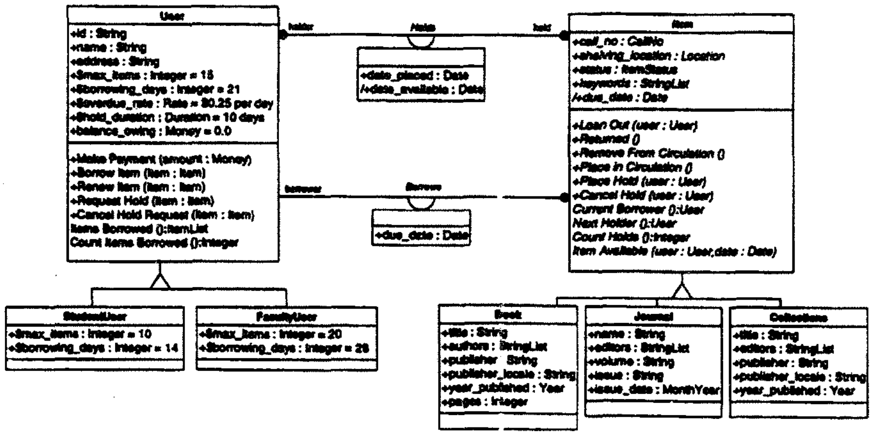

User
+id : String
+name : String
+address : String
+$max_items : Integer = 15
+$borrowing_days : Integer = 21
+$overdue_rate : Rate = $0.25 per day
+$hold_duration : Duration = 10 days
+balance_owing : Money = 0.0
+Make Payment (amount : Money)
+Borrow Item (item : Item)
+Renew Item (item : Item)
+Request Hold (item : Item)
+Cancel Hold Request (item : Item)
Items Borrowed ():ItemList
Count Items Borrowed ():Integer
holder
Holds
+date_placed : Date
/+date_available : Date
held
borrower
Borrows
+due_date : Date
Item
+call_no : CallNo
+shelving_location : Location
+status : ItemStatus
+keywords : StringList
/+due_date : Date
+Loan Out (user : User)
+Returned ()
+Remove From Circulation ()
+Place in Circulation ()
+Place Hold (user : User)
+Cancel Hold (user : User)
Current Borrower ():User
Next Holder ():User
Count Holds ():Integer
Item Available (user : User,date : Date)
StudentUser
+$max_items : Integer = 10
+$borrowing_days : Integer = 14
FacultyUser
+$max_items : Integer = 20
+$borrowing_days : Integer = 28
Book
+title : String
+authors : StringList
+publisher String
+publisher_locale : String
+year_published : Year
+pages : Integer
Journal
+name : String
+editors : StringList
+volume : String
+issue : String
+issue_date : MonthYear
Collections
+title : String
+editors : StringList
+publisher : String
+publisher_locale : String
+year_published : Year

Figure 1: Object Model for a Library System

of them in this paper, again due to space limitations.

2.1.1 Object Model

Variable A variable is a named placeholder which may assume values of a certain type or references to values of a type. A variable may be initialized during modeling.

ReferenceType ::= *ptr* | *val*

Variable
name : *Identifier*
type : *Type*
inital : *Value*
reftype : *ReferenceType*

inital matches *type*

Type denotes a collection of simple types such as integers, reals, characters and strings. *Value* denotes values of these types with minimum and maximum limitations imposed by implementation constraints. OMT does not have any type or value domain associated with it. Therefore, designers can choose any type or value for the variables in the design. For the ease of verification, we have chosen those types which are normally supported by object-oriented programming languages. The reason for this selection is to support verification of code generation facilities provided by the design tool. Defining the semantics of type and value domains is beyond the scope of our current work. We therefore took care of verification based on types and values in the implementation of the semantics.

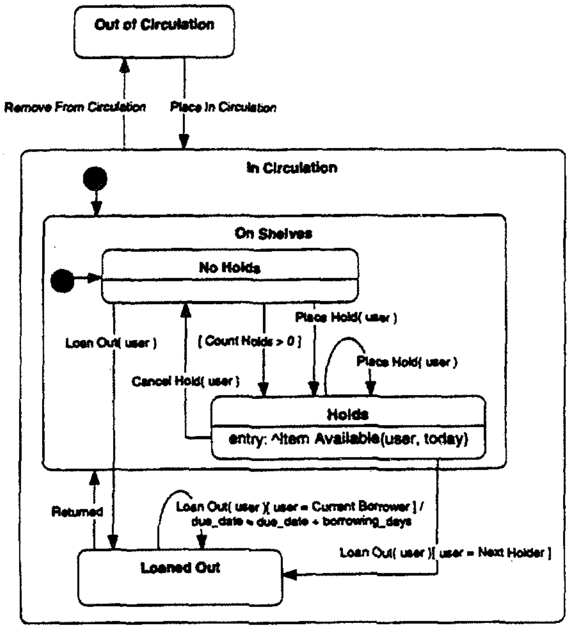


Figure 2: State Diagram for the class *Book*

Attribute An attribute is a variable, but it is attached to a class.

Attribute

Variable

$classid : ClassID$

$classtype(classid) = type \Rightarrow reftype = ptr$

If the type of the attribute is the same as the class in which the attribute is defined, the reference type of the attribute must be a pointer. Otherwise, the semantics of the attribute cannot be defined.

Operation An operation, also called *method* in this paper, is attached to a class. It consists of a signature (denoted collectively by the name of the operation, names and types of its parameters and the type of return value), names and types of local variables and the body (denoted by a set of executable statements).

$$
\begin{array}{l}
\textit{Operation} \\
\hline
name : Identifier \\
classid : ClassID \\
type : Type \\
parms : \mathbb{P}\ Parameter \\
locals : \mathbb{P}\ Local \\
body : \mathbb{P}\ ExecStmt \\
is\_abstract : YesNo \\
\hline
\langle \{name\}, names\_of\_parms(parms), names\_of\_locals(locals) \rangle \\
\quad \textsf{partition}\ \{name\} \cup names\_of\_parms(parms) \cup names\_of\_locals(locals) \\
body\ \textsf{stmtsuse}\ (names\_of\_parms(parms) \cup names\_of\_locals(locals)) \\
is\_abstract = yes \Rightarrow (locals = \varnothing \land body = \varnothing) \\
(\forall p : Parameter \mid p \in parms \bullet p.opname = name) \\
(\forall l : Local \mid l \in locals \bullet l.opname = name) \\
\hline
\end{array}
$$

The type *ExecStmt* used in *Operation* is defined as a basic type in [1]. This is because the executable statements of an operation are generally not defined during analysis and design. The variable *is_abstract* is of type *YesNo* and indicates whether or not the operation is intended to be abstract (not directly executable). Even though the body of the operation is not defined, the designer may deliberately choose certain operations to be abstract. The incomplete definition of an abstract operation is expected to be completed in one of the subclasses of the class in which the abstract operation is defined (e.g., the operation *draw* in the class *Polygon* is an abstract operation). The types *Parameter* and *Local* are defined to be the same as *Variable* with an additional component to indicate the operation to which the parameter or local variable is attached.

The first predicate in *Operation* indicates that the name of the operation, the names of parameters and the names of local variables to the operation must all be distinct. The binary infix relation **stmtsuse** has been specified in [1]. In this context, it asserts that the body of the operation uses the parameters and local variables. Since the body is not yet defined, it is not possible to implement this assertion. We therefore generate a warning message to the designer to indicate that the code for the operation must be written in such a way that it uses all the parameters and local variables defined in the operation.

Class A class has a structure defined by a set of attributes. In addition, a class also has a set of operations which describe the behavior of objects that are instantiated from this class. We use the term 'features' to denote a subset of attributes and operations together, as used by Eiffel [9]. The class's interface is defined by a subset of its features; this subset constitutes the exported features from this class.

Class

$$\begin{array}{|l}
\mathit{classid} : \mathit{ClassID} \\
\mathit{name} : \mathit{Identifier} \\
\mathit{attrs} : \mathbb{P}\, \mathit{Attribute} \\
\mathit{ops} : \mathbb{P}\, \mathit{Operation} \\
\mathit{expattrs} : \mathbb{P}\, \mathit{Attribute} \\
\mathit{expops} : \mathbb{P}\, \mathit{Operation} \\
\mathit{candidate\_key} : \mathrm{iseq}\, \mathit{Attribute} \\
\mathit{is\_abstract} : \mathit{YesNo} \\
\mathit{subdiagrams} : \mathbb{P}\, \mathit{DiagramID} \\
\hline
\mathit{attrs} \neq \varnothing \lor \mathit{ops} \neq \varnothing \\
\langle \{\mathit{name}\}, \mathit{names\_of\_attrs}(\mathit{attrs}), \mathit{names\_of\_ops}(\mathit{ops}) \rangle \\
\qquad \text{partition } \{\mathit{name}\} \cup \mathit{names\_of\_attrs}(\mathit{attrs}) \cup \mathit{names\_of\_ops}(\mathit{ops}) \\
\mathit{is\_abstract} = \mathit{yes} \Rightarrow (\exists\, o : \mathit{Operation} \mid o \in \mathit{ops} \bullet o.\mathit{is\_abstract} = \mathit{yes}) \\
\mathit{expattrs} \subseteq \mathit{attrs} \land \mathit{expops} \subseteq \mathit{ops} \\
\mathrm{ran}\, \mathit{candidate\_key} \subseteq \mathit{attrs} \\
(\forall\, a : \mathit{Attribute} \mid a \in \mathit{attrs} \bullet a.\mathit{classid} = \mathit{classid}) \\
(\forall\, o : \mathit{Operation} \mid o \in \mathit{ops} \bullet o.\mathit{classid} = \mathit{classid}) \\
\hline
\end{array}$$

The component *candidate_key* is an ordered arrangement of a subset of attributes of the class; it is defined as an injective sequence and hence it does not contain any duplicates. This sequence of attributes is used to collectively identify and uniquely distinguish an object of the class from other objects of the same class. Such a selection procedure is supported by OMT. The variable *subdiagrams* represents the set of state diagrams which describe the dynamic behavior of objects instantiated from this class.

The variables *classid* and *name* denote two identifiers of the same class. The variable *name* indicates the user defined name of the class. In OMT, two classes in two different modules* can have the same name. The component *classid* represents an identifier of the class that is internally generated by the system. This identifier is unique with respect to an application; an application typically consists of one or more modules.

The constraint

$\mathit{attrs} \neq \varnothing \lor \mathit{ops} \neq \varnothing$

indicates that a class cannot be totally empty; it should have at least one attribute or at least one operation. The next predicate asserts that the name of the class, the names of attributes and operations of the class must all be distinct. Notice that it may be possible that a local variable of an operation for a class and an attribute of the same class may have the same name. We do not specify the scope rules for variables because OMT does not impose any such restrictions. These scope rules will be imposed only by the programming languages chosen for implementing the current design.

*A module is a collection of classes.

A class becomes an abstract class if at least one of its operations is abstract. The exported features of a class must be a subset of the features defined within the class. It is important to notice that the set of features of a class not only include those syntactically defined in the class definition but also include those which the class definition can access. For example, if the class inherits the definition of another class, then the inherited features will also be included in the set of features of the class. During verification, we dynamically compute all such included features.

We now specify the static relationships between classes.

Association Conceptually, an association is defined as a semantic dependency between two or more classes. In [14], it is mentioned that a binary association from a class c_1 to another class c_2 can be implemented through a variable in c_1 whose type is c_2. Since within a class, a variable is defined as an attribute, as a parameter to an operation or as a local variable to an operation, we define the semantic dependency which establishes the association through the variable connection. This is specified as follows:

$$
\begin{array}{|l}
\_\ \mathsf{associates}\ \_ : \mathit{Class} \leftrightarrow \mathit{Class} \\
\hline
\forall\, c_1, c_2 : \mathit{Class} \bullet c_1\ \mathsf{associates}\ c_2 \Leftrightarrow \\
\quad (\exists\, o : \mathit{Operation} \mid o \in c_1.\mathit{ops} \bullet \\
\qquad (\exists\, p : \mathit{Parameter} \mid p \in o.\mathit{parms} \bullet p.\mathit{type} = \mathit{classtype}(c_2.\mathit{classid})) \lor \\
\qquad (\exists\, l : \mathit{Local} \mid l \in o.\mathit{locals} \bullet l.\mathit{type} = \mathit{classtype}(c_2.\mathit{classid}))) \lor \\
\quad (\exists\, a : \mathit{Attribute} \mid a \in c_1.\mathit{attrs} \bullet a.\mathit{type} = \mathit{classtype}(c_2.\mathit{classid}))
\end{array}
$$

The binary relation associates asserts that a class c_1 is associated with a class c_2 if one of the following three components in c_1 has c_2 as its type: an attribute, a parameter of an operation or a local variable of an operation. Notice that this definition indicates unidirectional association from c_1 to c_2. However, the notation for association in OMT is a simple straight-line which does not indicate the direction. This problem has been corrected in UML which uses directed arrows to represent associations.

We next assert that associates is transitive.

$$\forall\, c_1, c_2, c_3 : \mathit{Class} \bullet c_1\ \mathsf{associates}\ c_2 \land c_2\ \mathsf{associates}\ c_3 \Rightarrow c_1\ \mathsf{associates}\ c_3$$

Notice that the definition of associates does not directly imply the transitivity of associations. It can only be justified based on the notion of *propagation* [14, page 60].

An assocaition in OMT is modeled using a distinct connector symbol. Designers generally use this connector to model associations that are explicit and can be derived from the problem description (such as the association *Borrows* in Figure 1). We now specify the semantics of the connector for association. This semantics uses the definition associates which confirms the semantic dependency between the classes connected by the symbol for association.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Association}}\\
\hline
\mathit{name} : \mathit{Identifier}\\
\mathit{assocs} : \mathit{Class} \leftrightarrow \mathit{Class}\\
\mathit{propagate\_properties} : \mathit{Identifier} \leftrightarrow (\mathit{Class} \times \mathit{Class})\\
\mathit{candidate\_keys} : \mathbb{P}(\mathrm{iseq}\, \mathit{Attribute})\\
\mathit{attrs} : \mathbb{P}\ \mathit{Variable}\\
\hline
(\forall c_1, c_2 : \mathit{Class} \mid (c_1, c_2) \in \mathit{assocs} \bullet c_1 \ \mathsf{associates}\ c_2)\\
(\#\mathit{assocs} > 1 \Rightarrow\\
\quad (\exists c : \mathit{Class} \mid c \in (\mathrm{dom}\, \mathit{assocs} \cup \mathrm{ran}\, \mathit{assocs}) \bullet\\
\qquad (\forall c_1 : \mathit{Class} \mid c_1 \in (\mathrm{dom}\, \mathit{assocs} \cup \mathrm{ran}\, \mathit{assocs} \setminus \{c\}) \bullet\\
\qquad\quad c_1 \ \mathsf{associates}\ c \vee c \ \mathsf{associates}\ c_1)))\\
\mathrm{ran}\, \mathit{propagate\_properties} \subset \mathit{assocs}\\
(\forall i : \mathit{Identifier};\ c_1, c_2 : \mathit{Class} \mid i \mapsto (c_1, c_2) \in \mathit{propagate\_properties} \bullet\\
\quad i \in \mathit{Features}(c_1) \wedge i \in \mathit{Features}(c_2))\\
\bigcup\{ \mathit{ck} : \mathrm{iseq}\, \mathit{Attribute} \mid \mathit{ck} \in \mathit{candidate\_keys} \bullet \mathrm{ran}\, \mathit{ck}\} \subseteq\\
\quad \bigcup\{c : \mathit{Class} \mid c \in (\mathrm{dom}\, \mathit{assocs} \cup \mathrm{ran}\, \mathit{assocs}) \bullet c.\mathit{attrs}\}\\
\{a : \mathit{Variable} \mid a \in \mathit{attrs} \bullet a.\mathit{name}\} \cap\\
\quad \bigcup\{c : \mathit{Class} \mid c \in (\mathrm{dom}\, \mathit{assocs} \cup \mathrm{ran}\, \mathit{assocs}) \bullet \mathit{Features}(c)\} = \varnothing\\
\hline
\end{array}
$$

The variable *assocs* in *Association* denotes a set of ordered pairs of classes (c_1, c_2) where c_1 is associated with c_2. Notice that the ordered pair (c_1, c_2) represents a unidirectional association. A bidirectional association between c_1 and c_2 is represented by the two ordered pairs (c_1, c_2) and (c_2, c_1). Such a bidirectional association can be modeled using a single connector in OMT. A class may be associated with itself. In addition to capturing the semantics of self, unidirectional and bidirectional associations, the schema *Association* also captures the semantics of n-ary associations. When there are more than two classes participating in an association A, there must exist at least one class participating in A which is related to (associated in either direction) to all other classes participating in A. This condition enforces that multiple associations between disjoint sets of classes cannot be modeled using a single connector.

OMT uses the term *propagation* tc denote the automatic application of an operation to a collection of objects when the operation is invoked with one of the objects in that collection [14, pages 60]. From the design point of view, propagation indicates that the invocation of an operation on one class c influences the invocation of several operations in other classes which are semantically related to c. For example, if a class c_1 uses a local variable v of type c_2, c_2 being another class (c_1 is therefore associated with c_2), then c_1 is capable of invoking an operation *op* in c_2 (e.g., this is done by $v.op$ in C++). This indirectly means that the operation *op* is available in both c_1 and c_2. We say that *op* is propagated to c_1. In fact, attributes of c_2 are also propagated in a similar way. The variable *propagate_properties* in *Association* captures all such features that are propagated from the participating classes as a result of establishing an association. The function *Features* (not included in the paper, but specified in [1]) returns the names of attributes and operations of a class.

Given an association, a subset of attributes belonging to the participating classes can be used to identify a link of the association[†]. There is an explicit ordering among the subset of attributes selected to identify a link. For a given association, the designer may wish to identify more than one link and hence may assign more than one set of attributes. These sets of attributes are collectively specified as *candidate_keys* in *Association*. The ordering among the attributes is captured by the declaration 'iseq *Attribute*' which also indicates that no attribute is selected more than once to identify a link.

According to OMT, an association can also be modeled as a separate class with its own attributes. These attributes do not belong to any class participating in the association but are semantically related to all the participating classes. Such attributes are denoted by the variable *attrs* in the schema *Association*.

While specifying an association, we did not include multiplicity of an association[‡]. This is because there is no direct constraint imposed by multiplicity operators. When combined with other properties of an association, such as *Role* and *Qualifier*, a multiplicity operator may bring in additional semantic constraints. We have not included such constraints in this paper, but have specified them in [1].

Aggregation A class a is an aggregate of another class c (the latter being called as a component) if there is an attribute in a which is of type c. We also call c as a part of a. Below, we first define this relationship:

$$
\begin{array}{|l}
\_\ \mathsf{part\_of}\ \_ : \mathit{Class} \leftrightarrow \mathit{Class} \\
\hline
\forall c_1, c_2 : \mathit{Class} \bullet c_1\ \mathsf{part\_of}\ c_2 \Leftrightarrow \\
\quad\quad (\exists a : \mathit{Attribute} \mid a \in c_2.\mathit{attrs} \bullet \mathit{classtype}(c_1.\mathit{classid}) = a.\mathit{type})
\end{array}
$$

OMT supports recursive aggregation; in this case, the component inside the aggregate must be represented through a pointer. This condition is already captured in the definition of *Attribute*. The following constraint asserts that part_of is transitive:

$$
\begin{array}{l}
\forall c_1, c_2, c_3 : \mathit{Class} \mid c_1 \neq c_2 \wedge c_2 \neq c_3 \bullet \\
\quad\quad c_1\ \mathsf{part\_of}\ c_2 \wedge c_2\ \mathsf{part\_of}\ c_3 \Rightarrow c_1\ \mathsf{part\_of}\ c_3
\end{array}
$$

We next specify the type *Aggregation* that corresponds to the connector for aggregation in OMT.

[†] OMT uses the term *association* to refer to user-defined relationships between classes and the term *link* to denote the actual instances of the association established between objects of the participating classes.

[‡] In OMT, multiplicity of an association indicates the number of objects that can participate simultaneously in a link. For example, in Figure 1, the black circle in the associations *Holds* and *Borrows* stands for "zero or more". Consequently, the association *Holds* represents a many to many relationship between *User* and *Item* and *Borrows* represents a one-to-many relationship between the same pair of classes.

$$\begin{array}{|l}
\textit{Aggregation} \\
\hline
\textit{Association} \\
\textit{assembly} : \textit{Class} \\
\textit{component} : \textit{Class} \\
\hline
\textit{assocs} = \{(\textit{component}, \textit{assembly})\} \\
\textit{component}\ \mathsf{part\_of}\ \textit{assembly} \\
\hline
\end{array}$$

From the specification for part_of and *Aggregation*, one can observe that an aggregation is an association, but it is restricted to only two classes.

Generalization A generalization relationship exists between a class c_1 (called *superclass*) and another class c_2 (called *subclass*) if c_2 is a refined or specialized version of c_1. OMT uses the term *inheritance* to denote the mechanism by which a generalization relationship can be implemented. Accordingly, when a subclass inherits a superclass, the exported features of the superclass are inherited by the subclass. The subclass has the freedom to rename or to redefine any of the inherited features. Like association and aggregation, we first define the meaning of inheritance.

$$\begin{array}{|l}
\_\,\mathsf{inherits}\,\_ : \textit{Class} \leftrightarrow \textit{Class} \\
\hline
\forall c_1, c_2 : \textit{Class} \bullet c_1\ \mathsf{inherits}\ c_2 \Leftrightarrow \\
\quad c_1 \neq c_2 \wedge \\
\quad \textit{ExportedFeatures}(c_2) \subseteq \textit{Features}(c_1) \wedge \\
\quad \textit{ExportedFeatures}(c_2) \cap \textit{Features}(c_1) \neq \varnothing \wedge \\
\quad (\textit{Features}(c_2) \subset \textit{Features}(c_1) \vee \\
\qquad (\exists o_1, o_2 : \textit{Operation} \mid \\
\qquad\quad o_1 \in c_1.\textit{ops} \wedge o_2 \in c_2.\textit{expops} \wedge o_1.\textit{name} = o_2.\textit{name} \bullet \\
\qquad\quad o_1.\textit{parms} \neq o_2.\textit{parms} \vee o_1.\textit{type} \neq o_2.\textit{type} \vee o_1.\textit{body} \neq o_2.\textit{body}) \vee \\
\qquad (\exists a_1, a_2 : \textit{Attribute} \mid a_1 \in c_1.\textit{attrs} \wedge a_2 \in c_2.\textit{attrs} \wedge a_1.\textit{name} = a_2.\textit{name} \bullet \\
\qquad\quad a_1.\textit{type} \neq a_2.\textit{type}))
\end{array}$$

The function *ExportedFeatures* returns the names of attributes and operations that are exported from a class. Informally, the specification inherits asserts the following: When a class c_1 inherits from a class c_2,

- c_1 and c_2 must not be identical; hence, a class cannot inherit itself;
- all the exported features of c_2 are inherited c_1;
- at least one exported feature from c_2 is retained as unmodified in c_1; if everything is modified, then c_1 will not be a subclass of c_2; rather, it will be a different class altogether;
- one or more of the following three conditions must be satisfied:

- c_1 adds more features to those inherited from c_2;
- c_1 redefines one of the inherited operations;
- c_1 redefines one of the inherited attributes (changing the type); at this stage, we do permit to change the type of an inherited attribute as long as the new type is compatible with the old type; however, compatibility checks have not been included in the verifier at present.

$$\begin{array}{l} \forall\, c_1, c_2, c_3 : \mathit{Class} \bullet \\ \quad (c_1 \ \mathsf{inherits}\ c_2 \wedge c_2 \ \mathsf{inherits}\ c_3 \Rightarrow c_1 \ \mathsf{inherits}\ c_3) \wedge \\ \quad (c_1 \ \mathsf{inherits}\ c_2 \Rightarrow \neg\, (c_2 \ \mathsf{inherits}\ c_1)) \end{array}$$

The two global constraints assert that (i) inheritance is transitive and (ii) cyclic inheritance is not permitted.

An interesting note is that if there is an association between the superclass and a class c other than the subclass, then the subclass inherits that association as well; i.e., a new association is indirectly established between the subclass and c. This is because a subclass object can be substituted for a superclass object at runtime through polymorphic substitutions. The following constraint asserts this fact:

$$\begin{array}{l} \forall\, sub, sup : \mathit{Class} \bullet sub \ \mathsf{inherits}\ sup \Rightarrow \\ \quad (\forall\, c : \mathit{Class} \bullet c \ \mathsf{associates}\ sup \Rightarrow c \ \mathsf{associates}\ sub) \wedge \\ \quad (\forall\, c : \mathit{Class} \bullet sup \ \mathsf{associates}\ c \Rightarrow sub \ \mathsf{associates}\ c) \end{array}$$

We now define the type *Generalization* corresponding to the connector symbol for modeling generalization in OMT.

$$\mathit{Membership} ::= \mathit{disjoint} \mid \mathit{overlap}$$

$$\begin{array}{|l} \textit{Generalization} \\ \hline property : \mathit{Identifier} \\ superclass : \mathit{Class} \\ subclasses : \mathbb{P}\ \mathit{Class} \\ membership : \mathit{Membership} \\ \hline \forall\, c : \mathit{Class} \mid c \in subclasses \bullet c \ \mathsf{inherits}\ superclass \\ \hline \end{array}$$

A generalization symbol can be used to connect one superclass with several subclasses at the same time. In addition, a designer may also specify the property by which the subclasses specialize the superclass; this property is optional in OMT. There are two types of generalization relationships supported by OMT: *disjoint* and *overlap.* A disjoint generalization means an object is a member of only one subclass in the generalization. An overlapping generalization allows an object to be a member of more than one subclass simultaneously.

Two generalizations involving the same superclass and disjoint sets of subclasses must be generalizations along a different property. This is captured by the following global axiom:

$$\begin{array}{l}
\forall\, Generalization_1;\ Generalization_2 \bullet \\
\quad superclass_1 = superclass_2 \land \\
\quad subclasses_1 \cap subclasses_2 = \varnothing \Rightarrow property_1 \neq property_2
\end{array}$$

When a subclass multiply inherits from two distinct superclasses, the exported features of the superclasses must be disjoint, and the two superclasses cannot both be subclasses of the same disjoint generalization. Formally,

$$\begin{array}{l}
\forall\, sub, sup_1, sup_2 : Class \mid sup_1 \neq sup_2 \land \\
\quad sub \mathrel{\mathsf{inherits}} sup_1 \land sub \mathrel{\mathsf{inherits}} sup_2 \bullet \\
\qquad ExportedFeatures(sup_1) \cap ExportedFeatures(sup_2) = \varnothing \land \\
\qquad \neg\, (\exists\, g : Generalization \mid sup_1 \in g.subclasses \land \\
\qquad\quad sup_2 \in g.subclasses \bullet g.membership = disjoint)
\end{array}$$

OMT uses the term *leaf class* to denote a concrete class (not an abstract class) which has no subclasses. This is specified by the prefix operator $\mathsf{Leaf}_{\mathit{Class}}$ as follows:

$$\begin{array}{|l}
\mathsf{Leaf}_{\mathit{Class}}\ \_ : \mathbb{P}\, Class \\
\hline
\forall\, c_1 : Class \bullet \mathsf{Leaf}_{\mathit{Class}}\ c_1 \Leftrightarrow \\
\qquad \neg\, (\exists\, c_2 : Class \bullet c_2 \mathrel{\mathsf{inherits}} c_1) \land c_1.is\_abstract = no
\end{array}$$

2.1.2 Dynamic Model

In this section, we specify the modeling notations used in the dynamic model of OMT.

Event Class Conceptually, an event is an instantaneous occurrence or signal that is significant to the behavior of the entire system being modeled. From a designer's point of view, an event synchronizes operations that belong to the same class or to different classes. For example, in a client-server relationship, an event may signal the completion of an operation and thereby may inform a client who is waiting for that operation to complete. In OMT, events are introduced in the model to synchronize various actions in the dynamic model. Like links and associations, events that exhibit similar characteristics are modeled by an event class.

$$EventAttribute == Variable$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ EventClass\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
name : Identifier \\
attributes : \mathbb{P}\, EventAttribute \\
subdiagram : \mathbb{P}\, DiagramID \\
\hline
\forall\, ea : EventAttribute \mid ea \in attributes \bullet ea.name \neq name \\
\hline
\end{array}$$

An event class has a unique name which identifies the type of events that are instantiated from the event class. The attributes of an event class characterize various instances of events instantiated from the event class. The variable *subdiagrams* in *EventClass* indicates the set of state diagrams in which at least one instance of the event class appears. There may exist generalization relationship between event classes. Consequently, one event class may inherit another event class. Association and aggregation are, however, not defined for event classes. We have not included the generalization relationship between event classes in this paper; it can be found in [1].

Action An action refers to an instantaneous operation or the process of creation and dispatch of an event. For brevity, we only state the type *Action* in this paper.

$$Action ::= op\langle\!\langle Operation \rangle\!\rangle \mid se\langle\!\langle SendEvent \rangle\!\rangle$$

The type *SendEvent* is modeled as a schema in [1] which contains an event class identifying the type of the event to be sent and the target class receiving the event.

Internal Action An internal action is an action performed on an object in a state in response to an event. The object on which the internal action is exerted does not change its state as a result of performing the internal action.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InternalAction}} \\
\hline
state : StateID \\
class : Class \\
event\_class : EventClass \\
actions : \mathrm{seq}\, Action \\
\hline
(\exists\, o : Operation \mid \langle op(o) \rangle \text{ in } actions \bullet o.classid = class.classid) \\
(\exists\, e : SendEvent \mid \langle se(e) \rangle \text{ in } actions \bullet \\
\qquad \neg\, (e.event\_class = event\_class \lor e.event\_class \ \mathsf{inherits\_from}_{\mathsf{E}}\ event\_class)) \\
\hline
\end{array}
$$

The variable *class* refers to the class of the object on which the internal action is exerted. *state* refers to the particular state of the object in which the internal action is invoked. In fact, *state* must belong to the state diagram corresponding to *class*. This correspondence is not established in the schema *InternalAction*, but is captured in the specification of state diagrams (described later). The variable *event_class* refers to the type of event to which the internal action responds. *actions* denotes a sequence of actions describing the tasks to be performed by the internal action. The predicate part of the schema *InternalAction* asserts that *actions* can be divided into two categories: *operations* and *events* (as indicated by the type *Action*). If an action belonging to the internal actior is an operation, the class to which the operation belongs must be the same class to which the internal action is associated with. If the action is an event, then the event class from which the event originates should not be the event class associated with the internal action. Otherwise, an infinite loop will occur in the implementation of the internal action.

Activity According to Rumbaugh [14], an *activity* is different from an action; the former consumes some significant amount of time for execution while the latter is assumed to be performed in negligible time. At the design level, one might not know the anticipated execution time of an activity. However, the designer may want to give a hint about the duration of the activity to the programmer. For this reason, OMT syntactically distinguishes an activity from an action. An activity is associated with a state while an action is coupled with an event. There are two types of activities supported by OMT: *sequential* and *continuous*. Except for the annotation, these two types are not further elaborated in OMT [14]. Below, we give the specification for an activity; the specification is self-explanatory.

ActivityType ::= *sequential* | *continuous*

Activity

operation : *Operation*
state : *StateID*
class : *Class*
act_type : *ActivityType*
subdiagram : *DiagramID*

class.classid = *operation.classid*

State Semantically, the state of an object is described by the collection of values possessed by the attributes of the object at a given instant of time. Since OMT does not include the value domain, it is impossible to validate a state of an object based on the values of the object's attributes. We therefore validate only the representation of a state which is created using OMT notations.

State

$$\begin{array}{l}
sid : StateID \\
class : Class \\
diagram : DiagramID \\
name : Identifier \\
condition : Condition \\
entry\_actions : \mathrm{seq}\, Action \\
exit\_actions : \mathrm{seq}\, Action \\
activities : \mathrm{seq}\, Activity \\
internal\_actions : \mathrm{seq}\, InternalAction \\
is\_initial : YesNo \\
is\_final : YesNo \\
subdiagrams : \mathbb{P}\, DiagramID \\
\hline
diagram \in class.subdiagrams \land diagram \notin subdiagrams \\
(\forall\, o : Operation \mid \langle op(o) \rangle \ \mathrm{in}\ entry\_actions \frown exit\_actions \bullet \\
\quad o.classid = class.classid) \\
(\forall\, a : Activity \mid \langle a \rangle \ \mathrm{in}\ activities \bullet a.class = class) \\
(\forall\, ia : InternalAction \mid \langle ia \rangle \ \mathrm{in}\ internal\_actions \bullet ia.class = class) \\
is\_initial = yes \Rightarrow entry\_actions = \langle\rangle \\
is\_final = yes \Rightarrow (exit\_actions = \langle\rangle \land activities = \langle\rangle \land internal\_actions = \langle\rangle) \\
\neg\, (is\_initial = yes \land is\_final = yes)
\end{array}$$

A state is uniquely identified by a state id which is generated by the system. It also has a name introduced by the designer. The internal id and the name of a state are synonymous to their respective counterparts in the definition *Class*. A state may have actions and activities coupled to it.

The variable *condition* in *State* refers to a boolean expression which must be satisfied immediately upon entering into the state. The type *Condition* typically refers to the set of all well-formed formulas in first order predicate logic. Since we cannot describe a type of well-formed formulas using the Z notation itself, we define *Condition* as a basic type in this paper; the verifier takes care of evaluating predicate expressions. For the purpose of checking the validity of a condition, we specify the following evaluation function:

$$evaluate : Condition \longrightarrow BooleanType$$

The Z notation does not support a boolean type. So, we defined an enumerated type called *BooleanType* which consists of the two constants *trueT* and *falseT*. This type does not have any relationship with the truth values in the meta-language of the Z notation; it is treated just like another enumerated type.

The variables *entry_actions* and *exit_actions* describe two separate sequences of actions which must be performed while entering the state and while leaving the state respectively. The boolean variables *is_initial* and *is_final* indicate whether the state is an initial state, a final state or an intermediate state (indicated by setting both variables to *no*). If the state is an initial state, there are no entry actions associated with the state. Similarly, if the state is a final state, then there are no exit actions, activities or internal actions. Other constraints in the predicate part of the schema *State* establish consistency among other components of the schema.

Transition A transition denotes a change from a source state to a target state. It is invoked as a result of an event occurrence or may be invoked automatically, though automatic invocation of transitions are not desirable in many occasions. A transition may be guarded with a condition; the transition will be fired only if the associated event occurs and the condition is satisfied. A transition may transcend across levels of nested states.
The component *event_class* in *Transition* shows the event associated with the transition and the component *guard* denotes the associated condition. The variable *actions* refers to the sequence of actions associated with the transition. These actions are performed as a result of invoking the transition. The variable *exits* refers to the sequence of states that change due to the invocation of this transition (notice that a transition may transcend across nested states). Similarly, the variable *enters* denotes the sequence of states which become the targets of the transition. For the purpose of understanding, we have separated out the *source* and *target* states of the transition from the sequences *exits* and *enters* so that *exists* and *enters* only refer to nested states. If *exits* (or *enters*) refers to a non-empty sequence of states, the exit actions (or entry actions) of the states in the sequence are concatenated together according to the ordering in which these states appear in the sequence. While performing the actions as a result of invoking this transition, the sequence of exit actions are performed first, followed by the sequence of actions associated with the transition (denoted by *actions*) and finally the sequence of entry actions. This condition imposes a proper exit and entry protocol when invoking a transition. The last predicate in the schema *Transition* asserts that the conditions associated with the source and target states should not contradict the condition of the transition.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transition}}\\
\hline
\textit{diagram} : \textit{DiagramID}\\
\textit{class} : \textit{Class}\\
\textit{event\_class} : \textit{EventClass}\\
\textit{source} : \textit{State}\\
\textit{target} : \textit{State}\\
\textit{guard} : \textit{Condition}\\
\textit{actions} : \mathrm{seq}\,\textit{Action}\\
\textit{exits} : \mathrm{iseq}\,\textit{State}\\
\textit{enters} : \mathrm{iseq}\,\textit{State}\\
\textit{actions\_when\_fired} : \mathrm{seq}\,\textit{Action}\\
\hline
(\forall\, o : \textit{Operation} \mid \langle op(o)\rangle \ \mathrm{in}\ \textit{actions} \bullet o.\textit{classid} = \textit{class}.\textit{classid})\\
\textit{source}.\textit{class} = \textit{target}.\textit{class} = \textit{class}\\
(\forall\, s_1, s_2 : \textit{State} \mid \langle s_1, s_2\rangle \ \mathrm{in}\ \textit{exits} \bullet\\
\qquad s_1.\textit{exit\_actions} \frown s_2.\textit{exit\_actions} \ \mathrm{in}\ \textit{actions\_when\_fired})\\
(\forall\, s_1, s_2 : \textit{State} \mid \langle s_1, s_2\rangle \ \mathrm{in}\ \textit{enters} \bullet\\
\qquad s_1.\textit{entry\_actions} \frown s_2.\textit{entry\_actions} \ \mathrm{in}\ \textit{actions\_when\_fired})\\
((\textit{last}(\textit{exits})).\textit{exit\_actions} \frown \textit{actions} \frown\\
\qquad (\textit{head}(\textit{enters})).\textit{entry\_actions} \ \mathrm{in}\ \textit{actions\_when\_fired})\\
\neg\ ((\textit{evaluate}(\textit{source}.\textit{condition}) = \textit{trueT} \land \textit{evaluate}(\textit{guard}) = \textit{falseF}) \lor\\
\ (\textit{evaluate}(\textit{target}.\textit{condition}) = \textit{trueT} \land \textit{evaluate}(\textit{guard}) = \textit{falseF}) \lor\\
\ (\textit{evaluate}(\textit{guard}) = \textit{trueT} \land \textit{evaluate}(\textit{source}.\textit{condition}) = \textit{falseF}) \lor\\
\ (\textit{evaluate}(\textit{guard}) = \textit{trueT} \land \textit{evaluate}(\textit{target}.\textit{condition}) = \textit{falseF}))\\
\hline
\end{array}
$$

Several constraints have been defined which limit the way in which the types of transitions can exit a state. First, a state may not have two outgoing transitions that fire on the same type of event. Secondly, a state may not have two automatic transitions in which the guards are always trueT. These two constraints are defined by the following global axiom:

$$
\begin{array}{|l}
\_\,\mathsf{kind\_of}_{\mathsf{E}}\,\_ : \textit{EventClass} \leftrightarrow \textit{EventClass}
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{State};\ \textit{Transition}_1;\ \textit{Transition}_2 \bullet\\
\quad \theta\textit{State} = \textit{source}_1 = \textit{source}_2 \land\\
\qquad (\{\textit{event\_class}_1\} \neq \varnothing[\textit{EventClass}] \neq \{\textit{event\_class}_2\} \Rightarrow\\
\qquad\quad (\neg\ \textit{event\_class}_1 \ \mathsf{kind\_of}_{\mathsf{E}}\ \textit{event\_class}_2)) \land\\
\qquad \neg\ (\exists\, \textit{ia} : \textit{InternalAction} \mid\\
\qquad\quad \langle \textit{ia}\rangle \ \mathrm{in}\ \textit{internal\_actions} \bullet\\
\qquad\qquad \textit{ia}.\textit{event\_class} \ \mathsf{kind\_of}_{\mathsf{E}}\ \textit{event\_class}_1 \lor\\
\qquad\qquad \textit{ia}.\textit{event\_class} \ \mathsf{kind\_of}_{\mathsf{E}}\ \textit{event\_class}_2) \land\\
\qquad (\{\textit{event\_class}_1\} = \varnothing[\textit{EventClass}] = \{\textit{event\_class}_2\} \Rightarrow\\
\qquad\quad \neg\ (\textit{evaluate}(\textit{guard}_1) = \textit{trueT} \Leftrightarrow \textit{evaluate}(\textit{guard}_2) = \textit{trueT}))
\end{array}
$$

Finally, a state with an automatic transition must perform an activity but not a continuous activity, an entry action, or an exit action. In addition, if the state has an unguarded automatic transition, then the state cannot have any other outgoing transitions. A state with an outgoing automatic transition cannot have an internal action.

$$
\begin{array}{l}
\forall State; Transition_1 \bullet \\
\quad \theta State = source_1 \land \\
\quad \{event\_class_1\} = \varnothing[EventClass] \Rightarrow \\
\quad\quad ((\exists Activity \mid \\
\quad\quad\quad \langle \theta Activity \rangle \text{ in } activities \bullet \\
\quad\quad\quad act\_type \neq continuous) \lor \\
\quad\quad\quad entry\_actions \frown exit\_actions \neq \langle\rangle) \land \\
\quad\quad (guard_1 \in \varnothing[Condition] \Rightarrow \\
\quad\quad\quad \neg\, (\exists Transition_2 \bullet \theta State = source_2)) \land \\
\quad\quad internal\_actions = \langle\rangle
\end{array}
$$

The specification in [1] includes several additional constraints on transitions. For example, a state may not have two outgoing transitions that fire on the same event. This constraint forces the state model to be deterministic. We have not included these constraints in this paper due to space limitations.

State Diagrams A state diagram contains a collection of states and transitions. Each state diagram belongs to exactly one class. The state diagram for a class captures all the states of an object instantiated from this class. A state diagram may also have concurrent subdiagrams.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StateDiagram}} \\
\hline
\mathit{diagram} : \mathit{DiagramID} \\
\mathit{class} : \mathit{Class} \\
\mathit{states} : \mathbb{P}\, \mathit{State} \\
\mathit{initial\_states} : \mathbb{P}\, \mathit{State} \\
\mathit{final\_states} : \mathbb{P}\, \mathit{State} \\
\mathit{transitions} : \mathbb{P}\, \mathit{Transition} \\
\mathit{subdiagrams} : \mathbb{P}\, \mathit{DiagramID} \\
\hline
(\forall s : \mathit{State} \bullet s \in \mathit{states} \Leftrightarrow s.\mathit{diagram} = \mathit{diagram}) \\
(\forall t : \mathit{Transition} \bullet t \in \mathit{transitions} \Leftrightarrow t.\mathit{diagram} = \mathit{diagram}) \\
\mathit{initial\_states} \subset \mathit{states} \\
(\forall i : \mathit{State} \mid i \in \mathit{initial\_states} \bullet i.\mathit{is\_initial} = \mathit{yes}) \\
\mathit{final\_states} \subset \mathit{states} \\
(\forall f : \mathit{State} \mid f \in \mathit{final\_states} \bullet f.\mathit{is\_final} = \mathit{yes}) \\
(\forall s : \mathit{State} \mid s \in \mathit{states} \setminus (\mathit{initial\_states} \cup \mathit{final\_states}) \bullet \\
\qquad s.\mathit{is\_initial} = \mathit{no} \wedge s.\mathit{is\_final} = \mathit{no}) \\
\mathit{diagram} \notin \mathit{subdiagrams} \\
(\mathit{subdiagrams} = \varnothing \Leftrightarrow \mathit{transitions} \neq \varnothing \wedge \mathit{states} \neq \varnothing) \\
(\mathit{subdiagrams} \neq \varnothing \Leftrightarrow \mathit{transitions} = \varnothing \wedge \mathit{states} = \varnothing) \\
\hline
\end{array}
$$

The constraints of the schema *StateDiagram* ensure the consistency of class and diagram ids and the validity of initial and final states. The final constraint declares that a state diagram contains either a set of concurrent subdiagrams or a set of states and transitions, but not both. The reasoning behind this is that a transition cannot span concurrent state diagrams; otherwise the diagrams would not be concurrent.

A state s_2 is reachable from another state s_1 if s_1 is the source of a transition and s_2 is the target of the same transition. The specification in [1] includes the definition for reachability and a few properties of this relationship such as transitivity. Even though there is no explicit modeling notation for reachability, its specification still helps in tracing the states in a state diagram during verification.

3 Implementation of the Semantics

We have used the tool CLIPS [5] to implement the axiomatic semantics described in the previous section. This tool was developed by NASA at its Johnson Space Center. CLIPS is a rule-based system that uses a dialect of LISP. It supports three types of knowledge representations: *ordered-facts* (expressed as tuples and relations), *structured-facts* (expressed as frames and templates) and objects (described using object-oriented approach). We have used only *ordered-facts* and *structured-facts* in our implementation. CLIPS uses the Rete matching algorithm [7] which implements a forward-chaining search strategy.

A rule in CLIPS contains two parts: left-hand-side and right-hand-side. The left-hand-side is represented by a pattern of symbols and the right-hand-side is represented by a network of patterns of symbols. If the left-hand-side of a rule matches with the input, the rule will be placed on an agenda. An agenda is a sequence of rules to be fired. The inference engine in CLIPS selects and fires rules from the agenda based on the priority of each rule. The priority may be based on the conflict resolution strategy implemented in CLIPS or it may be a user-defined priority (also called a *salience factor*).

3.1 Online and Offline Verification

The verifier supports both online and offline verification. During online verification, every step in the design process is verified immediately after the step is completed. For example, as soon as the designer enters the name of a class, the verifier attempts to check whether there exists another class with the same name in the current module. In most cases, a designer prefers offline verification mainly because it provides flexibility to change the design until the time of verification. In the verifier, online verification is limited to primitive checks such as checking duplication of names, checking multiple inconsistent relationships between the same pair of classes (aggregation and generalization) and so on.

3.2 Mapping Z Specification to CLIPS

In this section, we give a brief summary of how the Z specification describing the semantics of OMT is implemented using CLIPS. Detailed explanation of the implementation can be found in [1].

The global variables and types in the specification are implemented as ordered-facts in the knowledge-base of the verifier. The general form of an ordered-fact is given as

(fact-name $\texttt{field}_1$ $\texttt{filed}_2$ ...$\texttt{field}_n$)

For example,

```
(type INTEGER)
```

declares that INTEGER is a type.

Schemas, relations and functions are all implemented using templates. For example, the schema *Variable* described in the previous section is implemented as follows:

```
(deftemplate variable
   (slot name
      (type IDENTIFIER)
      (default ?NONE))
   (slot vtype
      (type TYPE)
      (default ?NONE))
   (slot initval
      (type VALUE)
      (default ?NONE))
   (slot reftype
      (type REFERENCETYPE)
      (default ?NONE)))
```

Constraints in the specification (global constraints, type invariants and predicates in schema definitions) are captured by defining rules, one for each constraint. As an example, the predicate part of the schema *Variable* is implemented by the rule:

```
   (defrule valid-variable
      (variable (name ?n)(vtype ?t)(initval ?i)(reftype ?r))
      (matches ?i ?t)
⇒
   (assert (valid-variable (name ?n)(vtype ?t)(initval ?i)(reftype ?r))))
```

The above rule asserts that a variable declaration is valid only if its initial value matches its type; the term **matches** has been defined elsewhere in the verifier.

There are two types of rules defined in the verifier: *acceptance rules* which accept a design element and update the knowledge-base and *rejection rules* which validate and reject a design element if it does not satisfy the semantics. Acceptance rules are generally used to include facts while rejection rules are used to verify the satisfiability of the semantics.

3.3 Features of the Verifier

Following are some of the significant features of the verifier:

- The verifier has more than 300 rules and is capable of capturing more than 100 explicit errors that are most commonly encountered in an object-oriented design.

- It captures all implicit and explicit constraints in the specification. The implicit constraints include type checking of variables in the design.

- Some of the non-trivial errors that are caught by the verifier include *indirect recursive inheritance*, *ambiguous features multiply defined*, *unreachable and unexitable states* and *inconsistencies while establishing relationships among classes and states.*

- The verifier also supports run-time checking. Using this feature, a designer can instantiate an object and check its dynamic behavior. The formal semantics for this part is still under development.

The verifier has been tested by checking the design of a library management system and that of a temperature control system given in [14].

4 Conclusions and Future Work

In this paper, we presented an axiomatic semantics for OMT using the Z specification language and the implementation of the semantics using a rule-based system. The verifier captures several design errors which existing tools such as OMTool and Rational Rose do not capture. In addition, the verifier also supports run-time checking using which a designer can instantiate an object and verify its run-time behavior.

Following are two important tasks to be performed in the immediate future as a continuation of this work:

- The semantics will be extended to include all the notations supported by UML.
- The input format for the verifier will be changed to the design format used by Rational Rose (version 4.0) so that the verifier can be integrated with Rose; the Rose tool is developed by Rational Software Corporation, the same organization which develops UML.

References

[1] W. Baluta, "Verification of Object-Oriented Designs", Masters Thesis, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada, Jan 1997.

[2] G. Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin Cummings Publishing Company Inc., 1994.

[3] G. Booch *et al.*, *The Unified Modeling Language for Object Oriented Development: Documentation Set Version 0.8*, Rational Software Corporation, 1995.

[4] G. Booch *et al.*, *The Unified Modeling Language for Object Oriented Development: Documentation Set Version 0.91 Addendum* Rational Software Corporation, 1996.

[5] *CLIPS 6.0 Reference Manual*, vol. 1–2, NASA, Johnson Space Center, 1993.

[6] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press Computing Series, 1990.

[7] C.L. Forgy, "Rete: A Fast Algorithm for the Many Pettern and Many Object Pattern Match Problem", *Artificial Intelligence*, Vol. 19, No. 1, Jan 1982, pp. 17–37.

[8] I. Jacobson, *Object-Oriented Software Engineering*, Addison Wesley Publishing Co., 1992.

[9] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall International Series in Computer Science, 1989.

[10] *OMTool: Reference Manual*, Martin Manelta, 1993.

[11] K. Periyasamy, "Enriching OMT with Verification Strategies", *Semantic Integration: Workshop at OOPSLA '95*, Austin TX, Oct 1995.

[12] *Rational Rose 3.0: User's Guide*, Rational Software Corporation, 1995.

[13] *Rational Rose 4.0: User's Guide*, Rational Software Corporation, 1997.

[14] J. Rumbaugh et al., *Object-oriented Modeling and Design*, Prentice-Hall Inc., Englewood NJ, 1991.

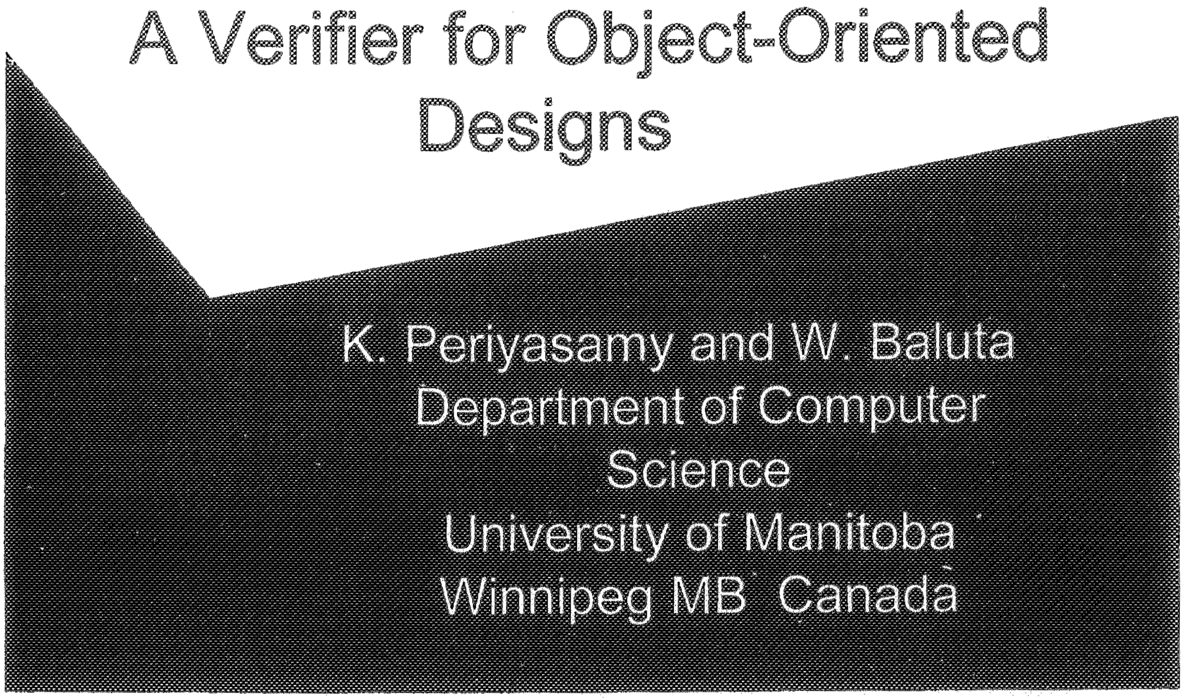
[15] J. M. Spivey, *The Z Notation: A Reference Manual*, Second Edition, Prentice Hall International Series in Computer Science, 1992.

[16] J.M. Spivey, *The fuzz Manual*, Computing Science Consultancy, 34 Westlands Grove, Stockton, York, UK, 1992.

Appendix – Supporting Templates for Class Verification Knowledge

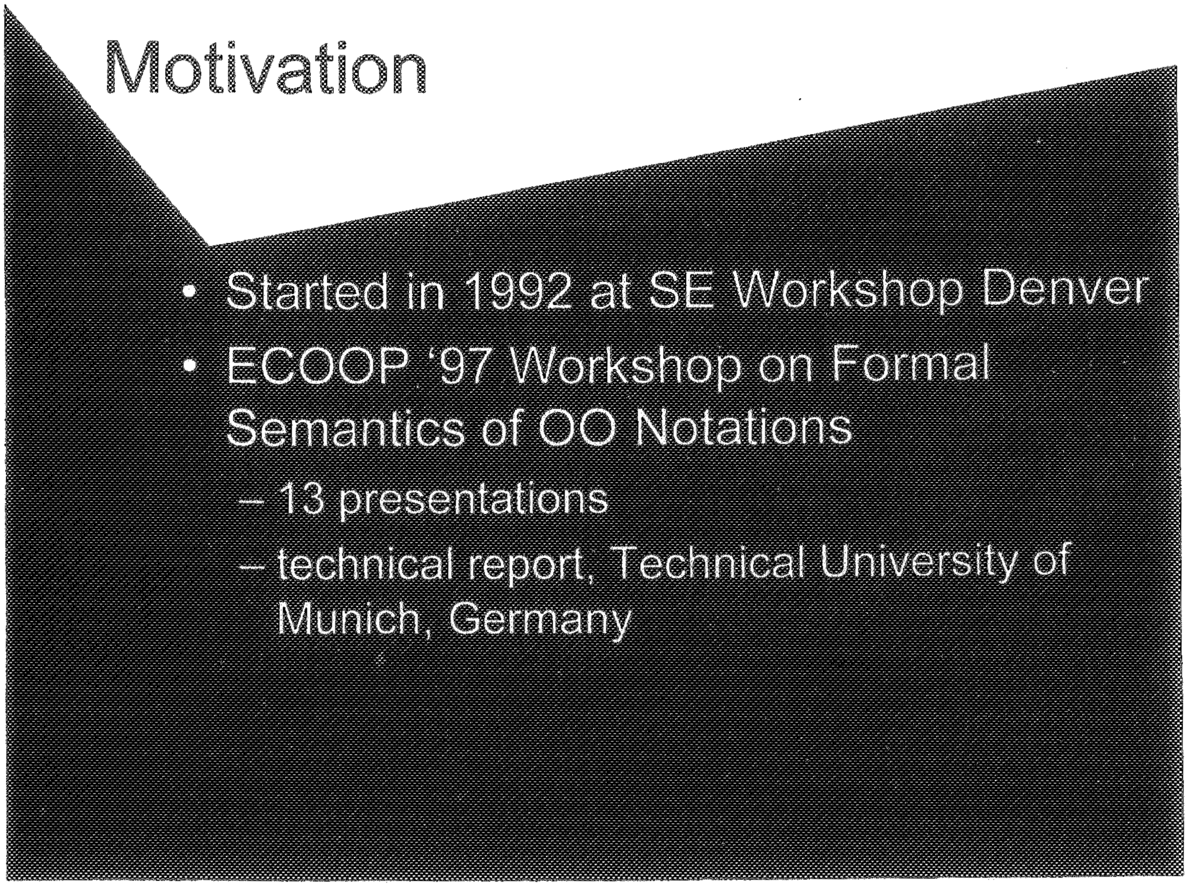
$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Class}} \\
\hline
classid : ClassID \\
name : Identifier \\
attrs : \mathbb{P}\ Attribute \\
ops : \mathbb{P}\ Operation \\
expattrs : \mathbb{P}\ Attribute \\
expops : \mathbb{P}\ Operation \\
candidate\_key : \mathrm{iseq}\ Attribute \\
is\_abstract : YesNo \\
subdiagrams : \mathbb{P}\ DiagramID \\
\hline
attrs \neq \varnothing \lor ops \neq \varnothing \\
\langle \{name\}, names\_of\_attrs(attrs), names\_of\_ops(ops) \rangle \\
\quad \mathrm{partition}\ \{name\} \cup names\_of\_attrs(attrs) \cup names\_of\_ops(ops) \\
is\_abstract = yes \Rightarrow \\
\quad (\exists\, o : Operation \mid o \in ops \bullet o.is\_abstract = yes) \\
expattrs \subseteq attrs \\
expops \subseteq ops \\
(\forall\, a : Attribute \mid \langle a \rangle\ \mathrm{in}\ candidate\_key \bullet a \in attrs \\
(\forall\, a : Attribute \mid a \in attrs \bullet a.classid = classid) \\
(\forall\, o : Operation \mid o \in ops \bullet o.classid = classid) \\
\hline
\end{array}
$$

```
(deftemplate Class
   (slot name
        (type SYMBOL)
        (default ?NONE))
    (multislot attrs)
    (multislot ops)
    (multislot expattrs)
    (multislot expops)
    (multislot candidate-key)
    (slot is-abstract
        (type SYMBOL)
        (allowed-symbols yes no)
        (default no))
    (multislot subdiagrams))

(defrule class-err-duplicate-attribute
    "The names of all a class' attributes must be distinct"
    (declare (salience -10))
    ?attr1 <- (Attribute (name ?a1) (cname ?c))
    ?attr2 <- (Attribute (name ?a2) (cname ?c))
    (test (and (eq ?a1 ?a2) (neq ?attr1 ?attr2)))
=>
   (printout t "OODV Error: The class " ?c
        " has a duplicated attribute " ?a1 "." crlf))

(defrule attributes-should-be-in-class-attrs
   "Attribute should be in the class' attribute list"
    (declare (salience 2))
    ?cls <- (Class (name ?c) (attrs $?alist))
    (Attribute (name ?a) (cname ?c))
    (test (not (member$ ?a $?alist)))
=>
    (modify ?cls (attrs (insert$ $?alist 1 ?a)))
    (printout t "OODV Update: Added attribute "
        ?c "." ?a
        " to attribute list."
        crlf))

(defrule class-err-duplicate-attribute-2
    "The names of all a class' attributes must be distinct"
    (declare (salience -10))
    (Class (name ?c)
        (attrs $?sublist1 ?a1 $?sublist2 ?a2 $?sublist3))
        (test (eq ?a1 ?a2))
    =>
        (printout t "OODV Error: The class "
            ?c
            " has a duplicated attribute " ?a1 "." crlf))
```



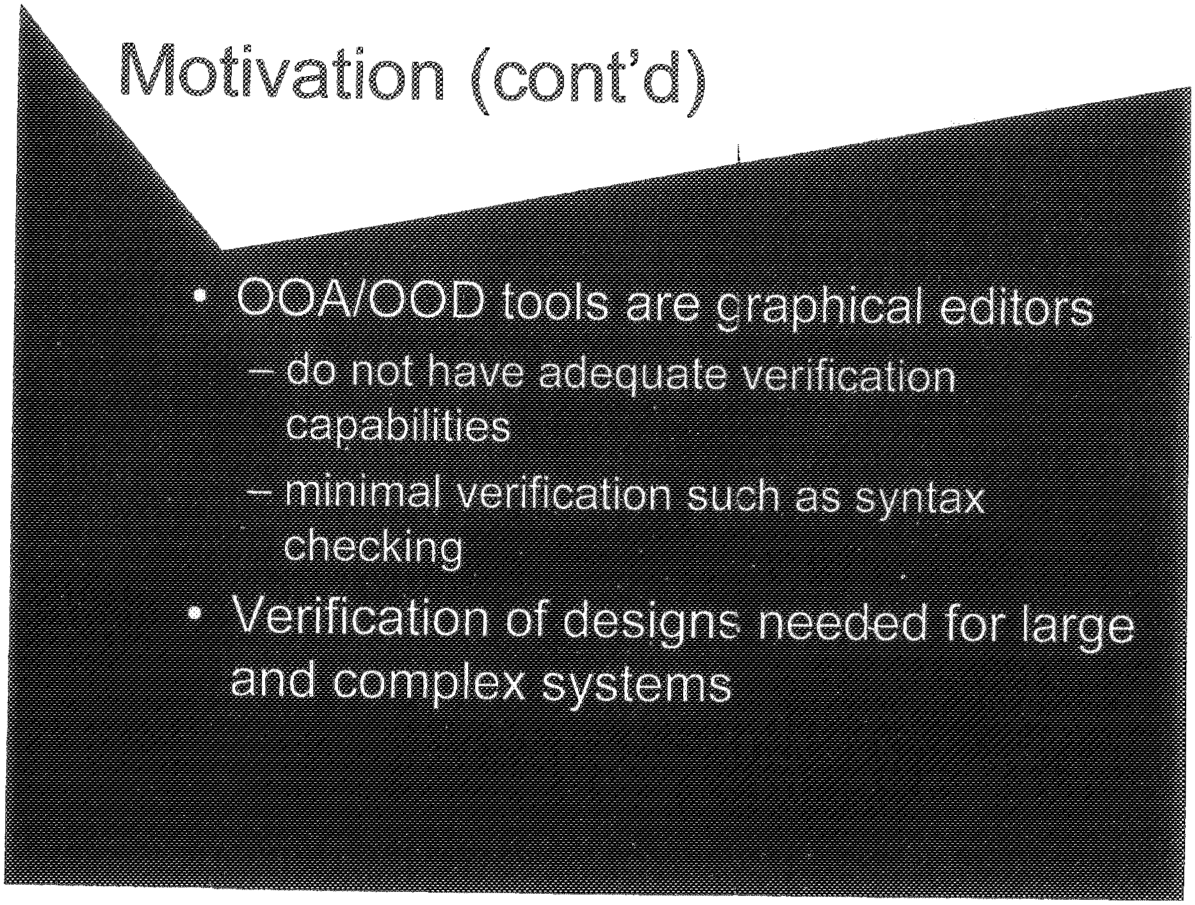
A Verifier for Object-Oriented Designs

K. Periyasamy and W. Baluta
Department of Computer Science
University of Manitoba
Winnipeg MB Canada



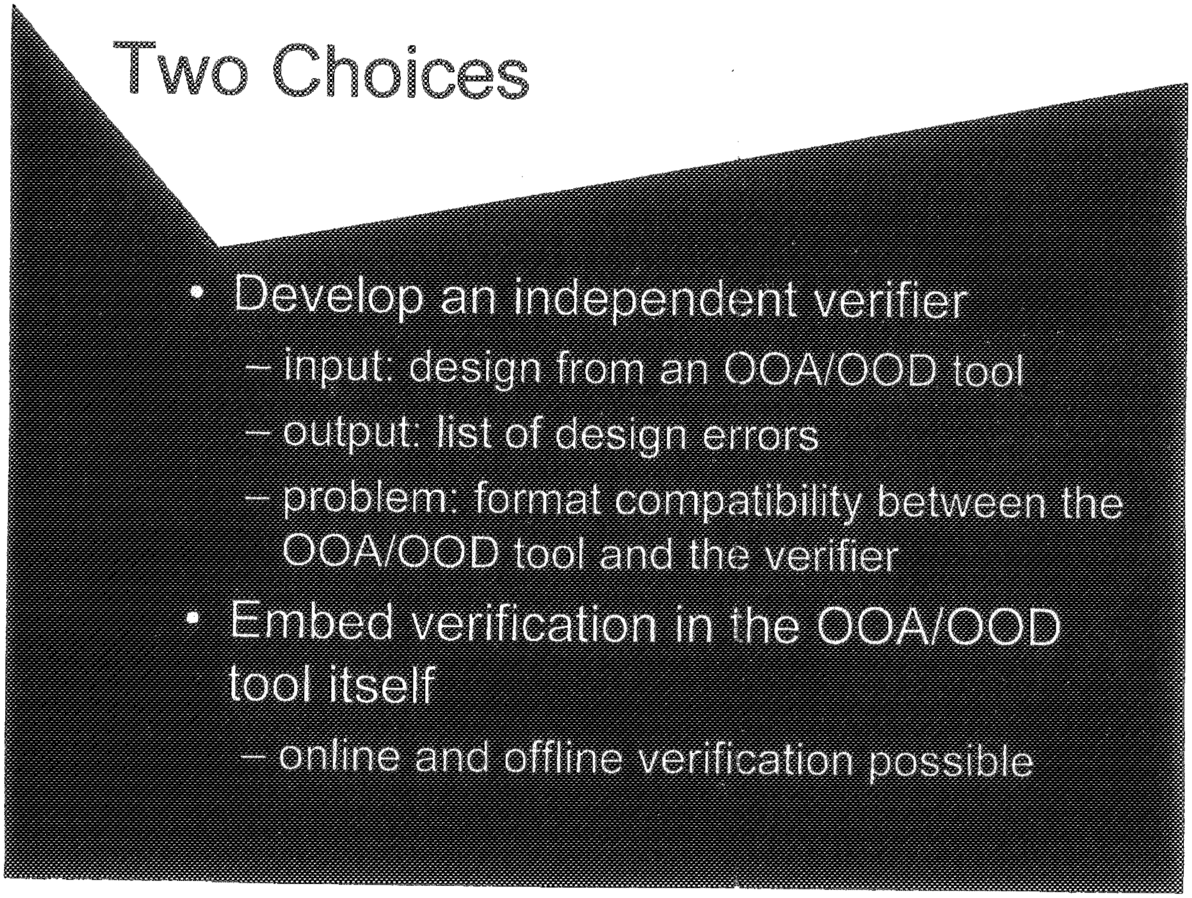
Motivation

- Started in 1992 at SE Workshop Denver
- ECOOP '97 Workshop on Formal Semantics of OO Notations
  - 13 presentations
  - technical report, Technical University of Munich, Germany



Motivation (cont'd)

- OOA/OOD tools are graphical editors
  - do not have adequate verification capabilities
  - minimal verification such as syntax checking
- Verification of designs needed for large and complex systems



Two Choices

- Develop an independent verifier
  - input: design from an OOA/OOD tool
  - output: list of design errors
  - problem: format compatibility between the OOA/OOD tool and the verifier
- Embed verification in the OOA/OOD tool itself
  - online and offline verification possible

Our Approach

- Hybrid approach:
  - we built a verifier which works offline
  - independent of OOA/OOD tools
  - plan to integrate with Rational Rose
  - plan to extend verification of designs based on UML notation

Verification and Validation

- Validation
  - through test cases
- Verification
  - checking correctness of development steps
  - requires a formalism
    - formal semantics of the notations used for OO design

Formal Methods in Industry

- IEEE Workshops
- IEEE Conference on Formal Engineering Methods, Japan 1998
- European Conferences and Workshops
- NASA Langley Centre's Workshop on Formal Methods

Current Status of Formal Methods in Industry

- Several formal methods
  - Z, VDM, Larch, OBJ3, ...
- Tool support
- Literature support
- Realization of the benefits in industrial projects

Our Work ...

- OMT notation for OO design
- Z notation for formal semantics
- implemented using CLIPS

Features Specified and Verified

- Class structure
- Relationships
  - aggregation
  - association
  - generalization
  - roles
- Event
  - event class
  - event generalization
- State
  - transitions
    - guarded, unguarded
  - nested
  - generalization
  - actions, activities

Static (Object) Model

- unique names
- abstract class must have at least one abstract operation
- consistency check for candidate keys
- cyclic inheritance not permitted
- multiple inheritance ambiguities checked
- recursive aggregation (referential)
- association (semantic dependency)

Dynamic Model

- Sequencing of actions and activities
- consistency checks on initial and final states
- consistency checks on transitions
  - cycles
  - automatic transitions
  - transition inheritance ...

Implementation of the Verifier

- CLIPS to implement the formal specifications
  - axioms implemented as rules
  - schemas implemented using templates
  - design information is converted into assertions (facts)
  - verification performed by checking consistency among rules and facts

About the Verifier

- Online verification
  - minimal; syntax and (somewhat) type checking
- Offline verification
  - all other aspects
- Functional Model not included
- Proprietary input format designed by Wasyl Baluta

About the Verifier (cont'd)

- Approx 40 pages of Z specification
- 300 rules for 100 explicit consistency checks
- Case Studies:
  - Library Management System
  - Examples in Rumbaugh's book (Object-Oriented Modeling and Design, 1991)

Continuing Work

- Update semantics to address UML notation
- Update implementation JESS (Java implementation of CLIPS)
- Integrate with Rational Rose
- Code generation (skeleton)
  - may work in conjunction with code generator in Rational Rose

360867

Demonstration of a Safety Analysis on a Complex System*

N. Leveson, L. Alfaro, C. Alvarado, M. Brown,
E.B. Hunt, M. Jaffe, S. Joslyn, D. Pinnel,
J. Reese, J. Samarziya, S. Sandys, A. Shaw, Z. Zabinsky

University of Washington
Seattle, WA 98195

For the past 17 years, Professor Leveson and her graduate students have been developing a theoretical foundation for safety in complex systems and building a methodology upon that foundation. The methodology (as described in her book *Safeware* [2]) includes special management structures and procedures, system hazard analyses, software hazard analysis, requirements modeling and analysis for completeness and safety, special software design techniques including the design of human-machine interaction, verification, operational feedback, and change analysis.

The *Safeware* methodology is based on system safety techniques that are extended to deal with software and human error. Automation is used to enhance our ability to cope with complex systems. Identification, classification, and evaluation of hazards is done using modeling and analysis. To be effective, the models and analysis tools must consider the hardware, software, and human components in these systems. They also need to include a variety of analysis techniques and orthogonal approaches: There exists no single safety analysis or evaluation technique that can handle all aspects of complex systems. Applying only one or two may make us feel satisfied, but will produce limited results.

We report here on a demonstration, performed as part of a contract with NASA Langley Research Center, of the Safeware methodology on the Center-TRACON Automation System (CTAS) portion of the air traffic control (ATC) system and procedures currently employed at the Dallas/Fort Worth (DFW) TRACON (Terminal Radar Approach CONtrol). CTAS is an automated system to assist controllers in handling arrival traffic in the DFW area.

Safety is a system property, not a component property, so our safety analysis considers the entire system and not simply the automated components. Because safety analysis of a complex system is an interdisciplinary effort, our team included system engineers, software engineers, human factors experts, and cognitive psychologists.

*This work was partially supported by grants from NASA Langley and NASA Ames.

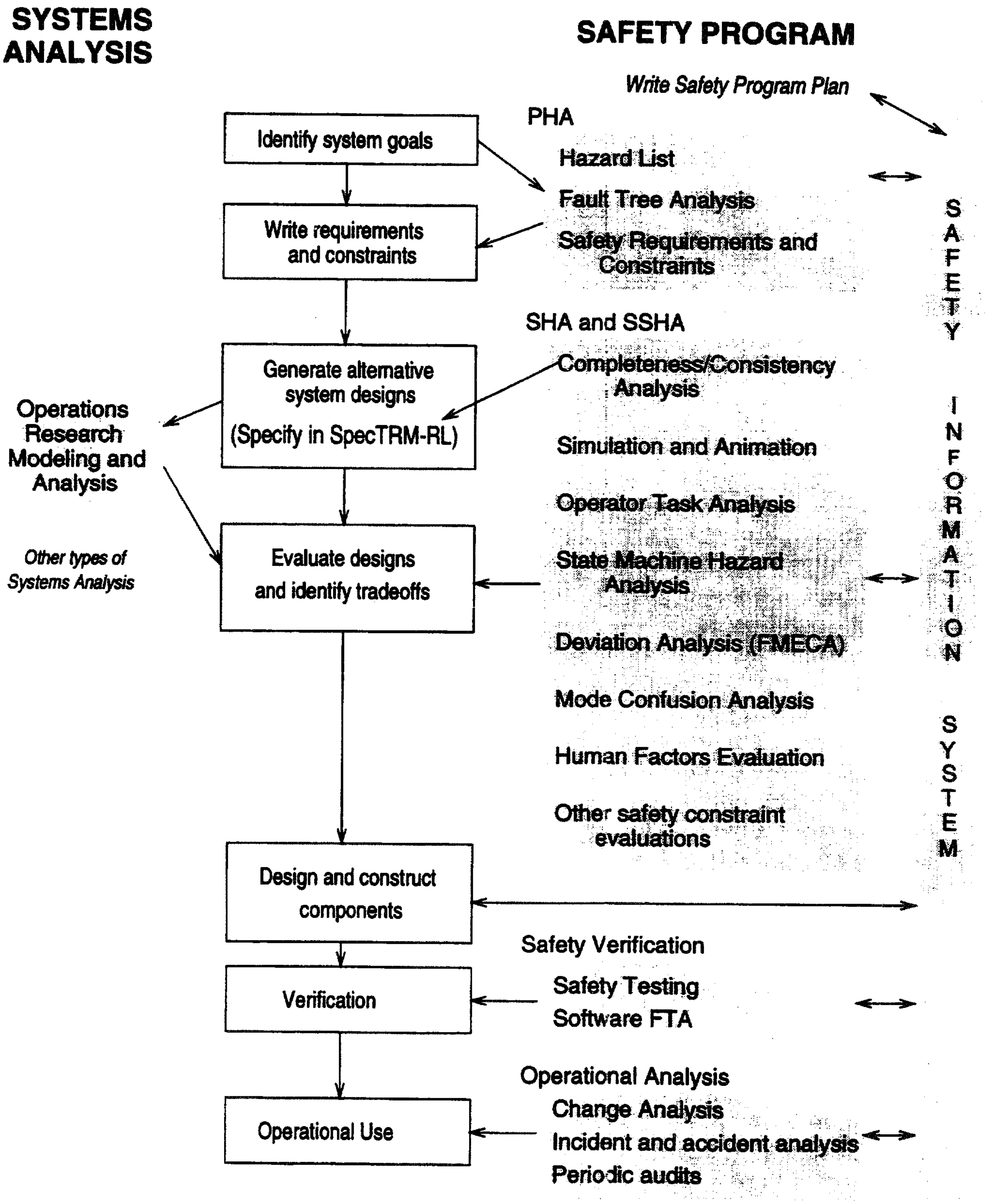

SYSTEMS ANALYSIS
SAFETY PROGRAM
Write Safety Program Plan
PHA
Identify system goals
Hazard List
Fault Tree Analysis
Write requirements and constraints
Safety Requirements and Constraints
SAFETY INFORMATION SYSTEM
SHA and SSHA
Generate alternative system designs
(Specify in SpecTRM-RL)
Completeness/Consistency Analysis
Operations Research Modeling and Analysis
Simulation and Animation
Operator Task Analysis
Other types of Systems Analysis
Evaluate designs and identify tradeoffs
State Machine Hazard Analysis
Deviation Analysis (FMECA)
Mode Confusion Analysis
Human Factors Evaluation
Other safety constraint evaluations
Design and construct components
Safety Verification
Verification
Safety Testing
Software FTA
Operational Analysis
Operational Use
Change Analysis
Incident and accident analysis
Periodic audits

Figure 1:

Figure 1 shows our design of a system safety program. Such a process is highly iterative and includes continual updating of what has been done previously as new information is gained through the system development process. In order to make the diagram less cluttered, the backward links are not shown, but note that the safety information system assists in the iteration process. An effective safety information has been found to rank second only to top management concern about safety in discriminating between safe and unsafe companies matched on other variables [1].

The center column of Figure 1 shows the standard system engineering tasks while the right column shows the special safety tasks and how they interact. We also performed some operations research modeling and analysis to demonstrate how information might be obtained and used to assist in making tradeoffs between alternative system designs.

We can only only provide an overview of the safety assessment and approach in this paper. Interested readers will find the final project report (containing many more details) at URL: http://www.cs.washington.edu/projects/safety/www/dfw. The assessment contained the following components:

Preliminary Hazard Identification and Standard Hazard Analyses. A safe system is one that is free from accidents or unacceptable losses. Accidents result from hazards, where a *hazard* is defined as a system state or set of conditions that can lead to an accident (given certain other, probably uncontrollable or unpredictable environmental conditions). In safety engineering, any safety assessment starts with identifying and analyzing the system for hazards. Once the hazards are identified, steps can be taken to eliminate them, reduce their likelihood, or mitigate their effects.

In addition, some hazard *causes* can be identified and eliminated or controlled. Although it is usually impossible to anticipate all potential causes of hazards, obtaining more information about them usually allows greater protection to be provided with fewer tradeoffs, especially if the hazards are identified early in the design phase. The hazards and the hazard causes can be used to write system safety requirements and constraints.

We performed a standard PHA using the DFW TRACON as an example, and wrote some preliminary safety requirements and constraints for CTAS and for air traffic control in general. As part of the PHA, we produced parts of a fault tree for TRACON operations that are related to the operation of CTAS. The information we generated was used in the demonstration of our analysis techniques.

Modeling. In order to do more than an evaluation of only the high-level ATC concept, a detailed specification or model of the behavior of the system components is required. A high-level design may appear to be safe while the detailed design contains hazardous component interactions. The hazards and design constraints identified in the first step must be traced to the system components, and assurance must be provided that the hazards have been eliminated or mitigated and the design constraints satisfied. Although theoretically this type of process could be performed on the detailed design of the system (including code if the component is a computer), the only

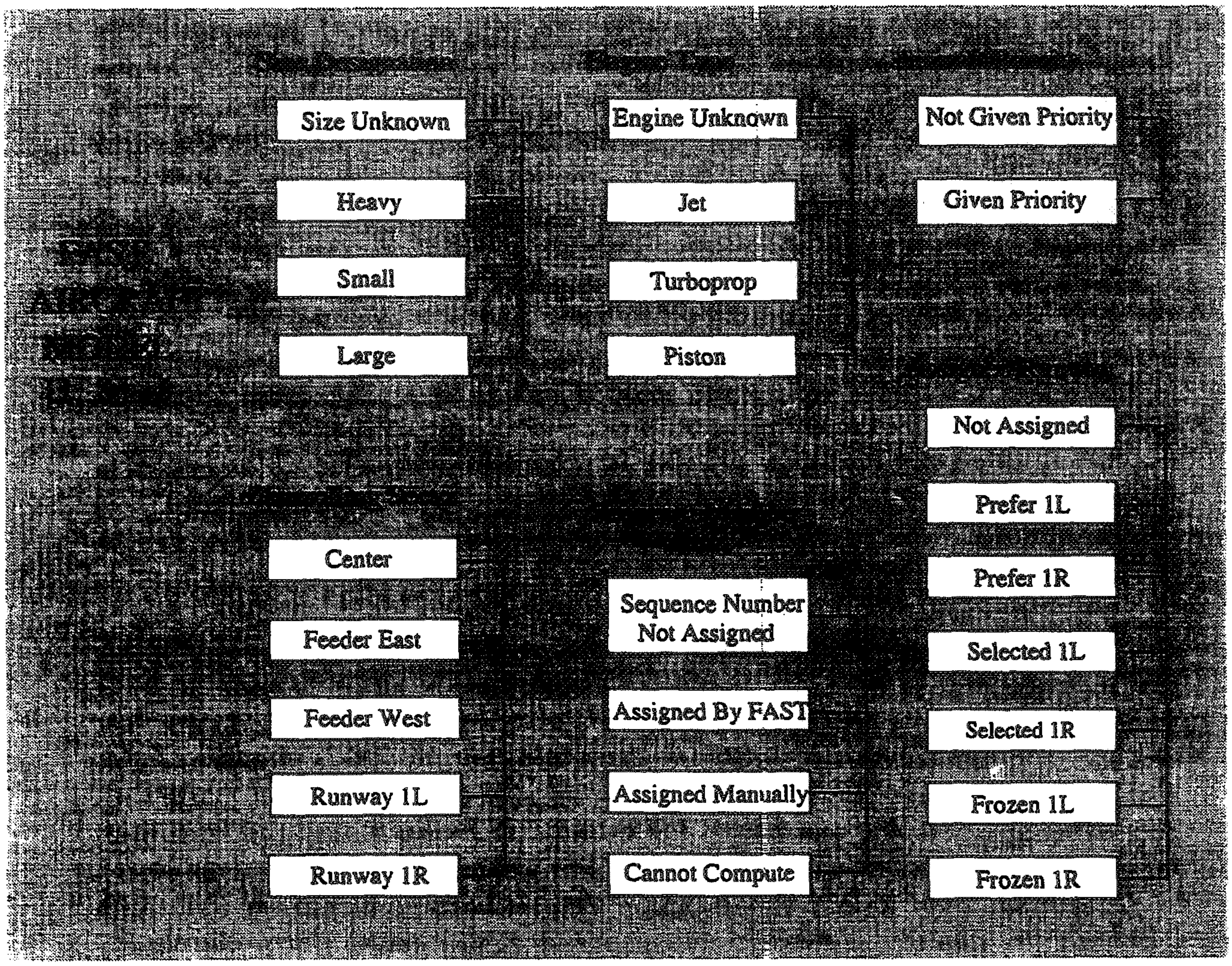


Figure 2:

practical approach is to provide hierarchical models and break the process up into steps. We built a state-based, blackbox model of the components of the DFW TRACON using a language called SpecTRM-RL that is readable and understandable with minimal training but has a formal foundation that allows analysis. Figure 2 shows a small piece of the model.

Simulation and Animation. Our models are executable and we have visualization tools to build animations appropriate to the model's domain (in this case, air traffic control). Our IB toolkit is an interface and visualization builder that allows users to build graphical user interfaces and animations of SpecTRM-RL models quickly and easily. Once the graphical design of the animation is completed, it can be attached to a SpecTRM-RL model to control the model's execution, display the execution outputs, or display internal states or actions of the model during execution.

As an example, we have created an animation that shows the behavior of aircraft within the TRACON area as the formal model is stepped through its states for a given set of inputs. This animation shows the controlled airspace and the aircraft in it, a timeline containing each aircraft's estimated time to landing, and an altitude indicator for the aircraft. As the model execution proceeds, the designer can see the aircraft moving along their projected flightpaths. During the simulated execution, the designer may click on parts of the animated display to get selected aircraft information such as speed, assigned runway, and assigned landing sequence number.

Controller Task Analysis. Humans form an important part of the ATC system, and they cannot be ignored in any safety analysis. We model operator procedures in the same language (SpecTRM-RL) as the other parts of the ATC system in order to allow executing and analyzing the ATC model as a whole. However, we use our visualization tools to display the information for human review in a more appropriate format. Figure 3 shows part of the task performed during a handoff procedure. This new language is used to display the nominal tasks that the controllers and pilots perform. We appreciate that humans do not necessarily perform tasks in the expected way. However, the first step in a safety analysis is to determine whether the nominal or expected behavior is safe. The implications of human error or deviations from nominal behavior is investigated in our other analyses. For the demonstration of the methodology, we built animations of the controller task models, including one that indicates through color coding the current cognitive and perceptual load on the pilot.

Completeness and Consistency Analysis. Accidents involving computers can usually be traced to incompleteness or other errors in the software requirements specification, not coding errors [3, 2]. Once a blackbox model of the required system behavior has been built, the model can be evaluated as to whether it satisfies design criteria that are known to minimize errors and accidents. We have developed such a set of criteria to identify missing, incorrect, and ambiguous requirements related to safety in process control systems. These criteria include much more than the mathematical completeness that is checkable on most formal models, although we can check this too.

State Machine Hazard Analysis. Hazard analysis techniques that use backward search start with a hazardous state and determine the events that could lead to this state. The analysis starts from hazards identified during the preliminary hazard analysis and identifies their precursors. The information derived about both normal and failure behavior can be used to redesign the system to prevent or minimize the probability of the hazard. We have found that the backward reachability graph explodes quickly for complex systems. Many of the branches are physically impossible or are less interesting than others, so we currently implement the process by having the analyst start the model in a hazardous state and work back one step at a time, using our backward simulation capability. At each step, the analyst prunes the tree of irrelevant branches and decides which branch to follow next.

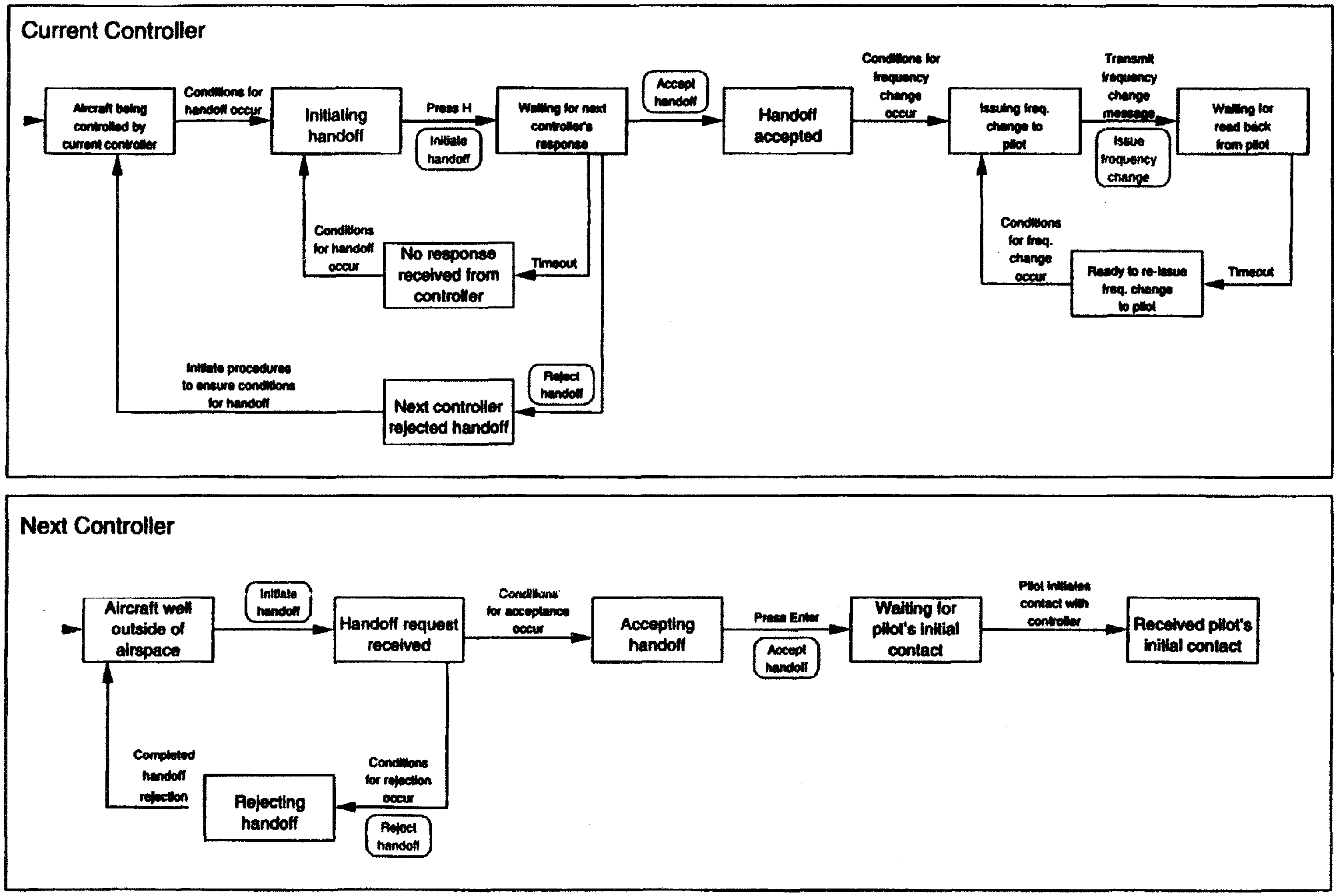

Current Controller
Aircraft being controlled by current controller
Conditions for handoff occur
Initiating handoff
Press H
Initiate handoff
Waiting for next controller's response
Accept handoff
Handoff accepted
Conditions for frequency change occur
Issuing freq. change to pilot
Transmit frequency change message
Issue frequency change
Waiting for read back from pilot
Conditions for handoff occur
No response received from controller
Timeout
Conditions for freq. change occur
Ready to re-issue freq. change to pilot
Timeout
Initiate procedures to ensure conditions for handoff
Next controller rejected handoff
Reject handoff
Next Controller
Aircraft well outside of airspace
Initiate handoff
Handoff request received
Conditions for acceptance occur
Accepting handoff
Press Enter
Accept handoff
Waiting for pilot's initial contact
Pilot initiates contact with controller
Received pilot's initial contact
Completed handoff rejection
Rejecting handoff
Conditions for rejection occur
Reject handoff

Figure 3:

Deviation Analysis. Forward search techniques start with an initiating event and trace it forward in time. Applying a Failure Modes and Effects Criticality Analysis (FMECA), HAZOP, or any other forward analysis technique to software is complicated by the large number of ways that computers can contribute to system hazards. In addition, when a forward analysis traces a failure to a computer component, it may be difficult to determine what affect the failure will have on the software behavior and outputs, particularly before the software has been implemented. We solve this problem using a new forward analysis technique for software called Software Deviation Analysis (SDA).

SDA is based on the underlying assumption that many accidents are the result of deviations in system variables. A deviation is the difference between the actual and correct values. SDA can determine whether a hazardous software behavior (usually an output) can result from a class of input deviations, such as measured aircraft speed too low (the measured or assumed speed is less than the actual speed). SDA is a way to evaluate system components for *robustness* (in the security community this is often called *survivability*) or how they will behave in an imperfect environment.

Human Error Analysis. Humans are and will continue to be for quite some time an important part of any air traffic control system. Therefore, an effective safety program cannot just look at the automated parts of the system but must consider the impact of human error on the system and the effect of system design on human error. Increased automation in complex systems has led to changes in the human controller's role and to new types of technology-induced human error. We approach this problem in two ways.

The first is a method we are developing for using our formal system models to detect error-prone automation features early in the development process while significant changes can still be made. We have taken what has been learned by cognitive psychologists from past accidents, incidents and simulator studies, and identified a set of automation design flaws that are likely to induce errors in humans that interact with the automation. The information produced from this *mode confusion analysis* can be used to redesign the automation to take out the error-inducing features or to design better human-machine interfaces, operator procedures, and training programs.

Our second approach to safety analysis of human error is a more traditional form of human factors analysis. For this DFW CTAS study, we first looked at the types of human errors in the current ATC system and then performed a comparative analysis of the controller's job before and after CTAS. Potential safety issues were identified involving decreased situation awareness, increased vigilance requirements, and skills degradation. Normally this step would be followed by running experiments to determine the effect of the changes on human performance with respect to these identified safety issues. However, the time limitations of this study did not allow us to perform this final step. Instead, we described some relevant hypotheses and an experimental paradigm for evaluating these hypotheses.

Operations Research Modeling and Analyses. Safety is not the only goal of an air traffic control system. The systems engineering process involves making tradeoffs between various goals such as safety, throughput, and fuel efficiency. If a proposed upgrade turns out to degrade safety significantly while providing only minimal benefit in terms of throughput or fuel economy, then it may not be worthwhile to implement it or an alternative design may provide a better result. We used a discrete-event simulation to compare the total delay and fuel burn for five different scheduling algorithms that may be used to control aircraft from arrival at an enroute ATC center to their arrival at a feeder gate into the TRACON.

The scheduling algorithms ranged from a basic algorithm that does not allow any passing or altitude overtakes and simply delays aircraft, to a more sophisticated scheduling algorithm that allows passing and introduces a freeze horizon. The scheduling algorithms may be viewed as having different levels of safety. For example, two of the five scheduling algorithms seek the path for each aircraft that minimizes fuel burn, even though this path may result in two or more advisories from the controller. The other three scheduling algorithms minimize the number of advisories issued to reduce the number of communications between controller and pilot and thereby minimize an important causal factor in accidents.

The models can provide information such as the amount of delay, or the amount of fuel consumed for various air traffic profiles operating under different scheduling algorithms. We showed how these models can be used for tradeoff studies, in order to evaluate proposed scheduling algorithms in CTAS.

Intent Specifications. The types of formal modeling and hazard analysis described so far provide a comprehensive assessment methodology. The most effective way to create a safe system, however, is to build safety in from the beginning. The preliminary hazard analysis should start at the earliest concept formation stages of system development and the information should be used to guide the emerging design. Later, system and subsystem hazard analysis information is used to evaluate the designs and make tradeoff decisions.

Intent specifications support both (1) general system development and evolution and (2) system safety analysis. The design rationale and other information that is normally lost during development are preserved in a single, logically structured document whose design is based on fundamental principles of human problem solving. Safety-related requirements and design constraints are traced from the highest levels down through system design, component design, and into hardware schematics or software code. An important feature of intent specifications is that they integrate formal and informal specifications and enhance their interaction.

We did not have the information necessary to build a complete intent specification for CTAS. Instead, we built a sample intent specification for TCAS, an airborne collision avoidance system with similar aircraft tracking functions. The sample TCAS system specification (800 pages long) can be viewed at the following URL: http://www.cs.washington.edu/research/projects/safety/www/intent.ps

Other Parts of a Complete Safety Program. We did not perform any safety testing or evaluation of the actual code, but this would obviously be part of any complete safety program. During operational use of the system, incident and accident data would be collected and analyzed along with analysis of any changes or modifications. Change analysis uses the same procedures as those used during the original development. Our modular models along with the tracing of safety-related constraints to the design and code that is part of an intent specification should make it easier to perform the change analysis. In addition, periodic audits should be made to ensure that the assumptions underlying the safety analysis (which are recorded in the intent specification) have not been violated by the natural changes that occur in any system over time.

Summary

How best to assure safety in complex systems is an open question. We have described one approach to achieving this goal that has been demonstrated on several real systems, including proposed ATC automation upgrades. Safety, however, is not something that is simply assessed after the fact but must be built into a system. By identifying safety-related requirements and design constraints early in the development process, special design and analysis techniques can be used throughout the system life cycle to guide safe software development and evolution.

References

[1] Urban Kjellan. Deviations and the Feedback Control of Accidents. in J. Rasmussen, K. Duncan, and J. Leplat (eds.) *New Technology and Human Error*, pp. 143-156, John Wiley & Sons, 1987.

[2] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Co., 1995.

[3] Robin R. Lutz. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. *International Symposium on Requirements Engineering*, San Diego, 1992.

Demonstration of a Safety Analysis on a Complex System

Nancy Leveson (CS) UW

Liliana Alfaro (IE) UW
Christine Alvarado (CS) Dartmouth
Molly Brown (CS) UW
Earl Hunt (Psych) UW
Matt Jaffe (CS) ERAU
Susan Joslyn (Psych) UW

Denise Pinnell (CS) UW
Jon Reese (CS) UW
Jeffrey Samarziya (CS) UW
Sean Sandys (CS) UW
Alan Shaw (CS) UW
Zelda Zabinsky (IE) UW

The Problem

- Two competing trends in ATC:
  - Want to increase throughput
  - Want to decrease accident rate
- Inevitable that will use more automation.
- Will safety increase or decrease?
  - Difficult to ensure software quality.
  - Shared control causes the most problems.
  - Error margins will decrease.
- Traditional approaches do not work well in complex systems.

Our Approach

- Safeware
  - Theoretical foundation
  - Complete methodology that spans life cycle
- SpecTRM: Set of integrated tools to build complex control systems.
- Requires a multidisciplinary team.

Preliminary Hazard Analysis

- Hazard list
- Fault Tree Analysis
- ATC Requirements and Design Constraints

Modeling Language

- Designed RSML for specifying TCAS II
  - Combined specification and modeling language
  - Both readable without much training and formally analyzable.
- Used this experience to design SpecTRM-RL
  - Enhanced reviewability and readability
  - Mode guidance in building models
  - Eliminated error-prone features

SpecTRM-RL

- Executable, black-box specifications
  - System components specified only in terms of outputs and the inputs that trigger them.
  - Specify external behavior only - no internal design.
  - System behavior is combined behavior of components.
- Underlying formal model - RSM
  (Requirements State Machine)

Operator Task Models

- To ensure safe and efficient operations, must look at the interaction between the human controllers and the computer.
- Use same underlying formal modeling language.
- Designed a visual representation more appropriate for the task modeling.
- Can be executed and analyzed along with other parts of the DFW TRACON model.

Safety Analysis

- Completeness and consistency analysis
- Simulation and animation
- Operator task analysis
- State machine hazard analysis
- Deviation Analysis

Safety Analysis (con't.)

- Mode Confusion Analysis
- Human Factors Evaluation
- Timing Analysis
- OR modeling and analysis

Intent Specifications

- Bridge between disciplines
- Support for human problem solving
- Traceability
- Support for safety analyses
- Integration of formal and informal specifications
- Assistance in software evolution
- Hierarchical abstraction based on "why" (design rationale) as well as what and how.

Conclusions

- Demonstrated the components of a safety analysis
- Safety needs to be built in and not assessed after-the-fact.
- Feasible to do for a complex system.
- Final report can be found at:

www.cs.washington.edu/research/projects/safety/www/dfw/final.ps
www.cs.washington.edu/homes/leveson

/

Automatic Generation of Test Coverage Analyzers

Sanjeev Aggarwal Utpal Bhattacharyya
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur 208016, INDIA
Email: ska@iitk.ernet.in

Abstract

Software Test Coverage Analyzers are very useful in software testing process. The development of Test Coverage Analyzers is a labor-intensive and time consuming process. This paper describes a generative approach for development of Test Coverage Analyzers, which can generate test coverage analyzer for a language, given the grammar and probe specification. Application experiences have shown that, a 150 lines (approximately) specification is enough for generating a Test Coverage Analyzer; and the productivity gain is as much as 25-35 times of the hand coded development.

1 Introduction

Software testing [Bei90] consumes nearly half of the effort required to produce a working software system. Due to this reason, software development is accompanied by quality assurance activity, and Software Testing is a critical element of software quality assurance. Software testing tools, that can reduce the testing time without reducing the effectiveness of testing, are very much valuable in this context. One of the very important tools used for testing, is Coverage Analyzer.

2 Test Coverage Analyzers

A Test Coverage Analyzer takes a source program as input, and inserts software probes into the source code, at certain places, dictated by some coverage measures [Cor96]. Using these software probes, it monitors the test run of the program and determines the coverage measures. The ideal places in a program for inserting probes, are the places, where transfer of control takes place (e.g branch statement, procedure call, logical conditions etc.), so that traversal of any basic path can be captured. However, optimal probe insertion techniques available in the literature [RKC75, Pro82, Aga94], may be used to avoid redundant probe insertion.

The basic task of a Test Coverage Analyzer(TCA) is to identify these places. To accomplish this task, it needs to parse the source program. Thus, the first requirement is the grammar specification of the source language.

The various phases of a test coverage analyzer perforn following activities (refer to Figure 1 for the functional diagram):

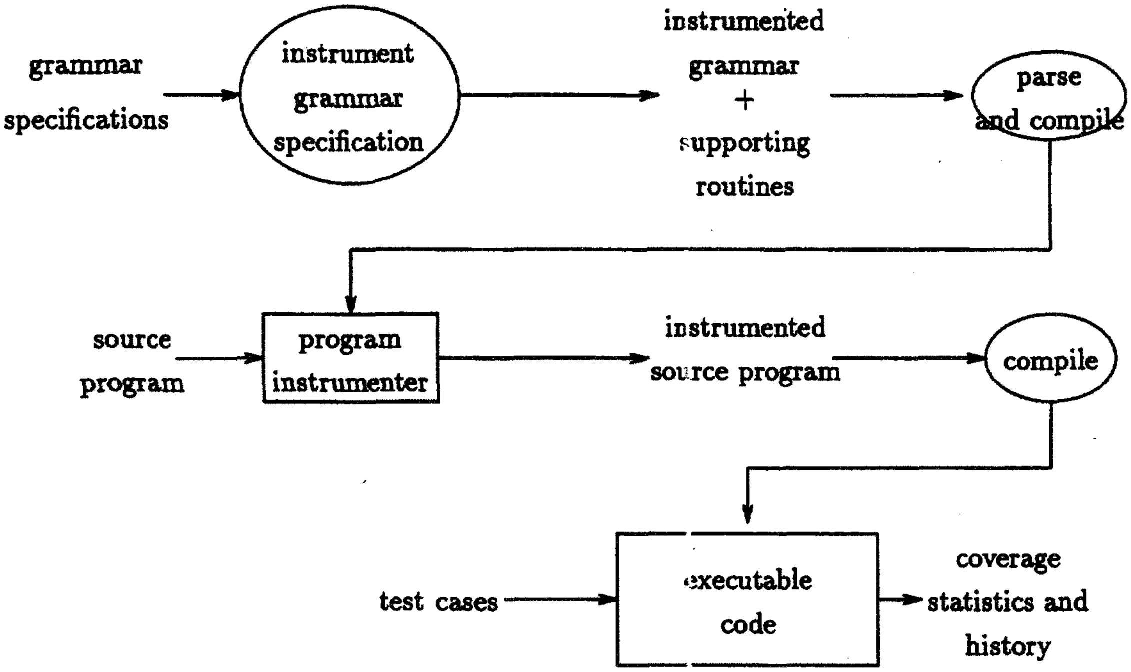

grammar specifications
instrument grammar specification
instrumented grammar + supporting routines
parse and compile
source program
program instrumenter
instrumented source program
compile
test cases
executable code
coverage statistics and history

Figure 1. Functional Diagram of Test Coverage Analyzer

- Identify productions in the grammar where probe actions are to be inserted.
- Put probe actions in the selected productions
- Write supporting modules for the probes and data structures

Normally, these steps are carried out manually. However, our experience with design of test coverage analyzers for Ada95 and C, shows that, this is the most crucial process and consumes the maximum amount of time in the development cycle. We assume that the reader is familiar with the development of Test Coverage Analyzers (TCA) hence we do not go into the details.

2.1 Program Instrumenter

Figure 2 shows the functional architecture of Program Instrumenter module. The basic components of this module are:

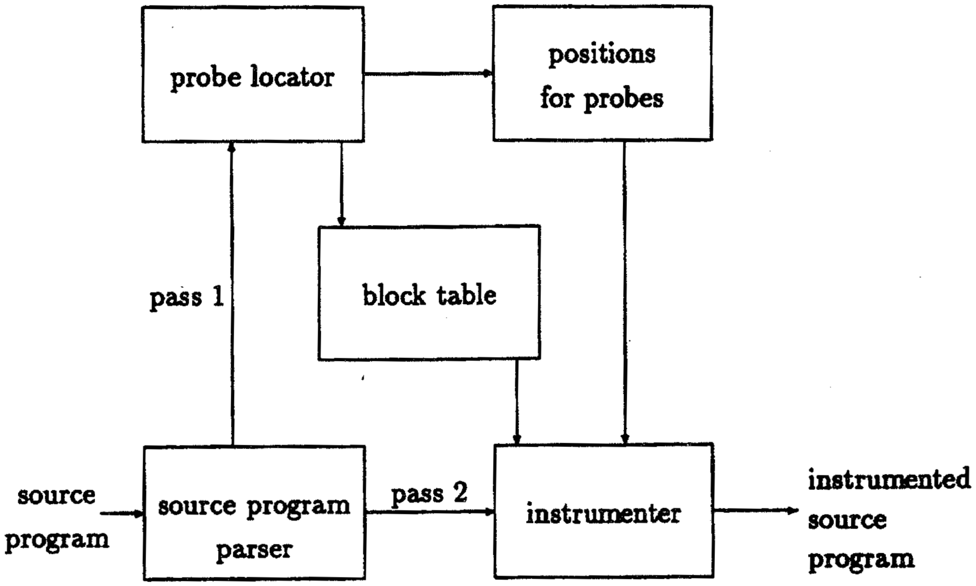


Figure 2: Functional Architexture of Program Instrumenter

Source Program Parser , reads the source program and provides information regarding the *beginning* and *ending* of the blocks, and the location of the probes to be inserted, to the *Probe Locator.*

Probe Locator constructs *block table*, and stores the position of the probes in a table. Structure of the block table is shown below:

Type of block	begin line number	end line number	tabs	depth

Structure of block table

Instrumenter inserts software probes in the source program, with the help of the block table and the table storing the probe locations.

A source program is instrumented with three types of probes:

- probes for monitoring the test run
- probes for coverage reporting, and
- probes for maintaining the coverage history, if multi-session testing is supported.

3 Generating Test Coverage Analyzer

Our experience shows that *instrumenting* the grammar is the most crucial phase in the development process, and is carried out manually. Thus, if this phase can be automated, fast development of TCAs can be achieved. Also, the quality of the software is better in case of generated softwares. These are the two factors that led us to a generative approach for Test Coverage Analyzers. The basic philosophy of our work is to ease the task of the developer, and to minimize programming from the development process. The tool that generates a Test Coverage Analyzer has been referred as Generic Test Coverage Analyzer(GTCA). Figure 3 shows the functional diagram of GTCA.

3.1 Design

The design phases of GTCA are same as TCA except a new phase called Grammar instrumenter. This phase automates the task of instrumenting the grammar. It consists of two components

Specification-parser takes input from a probe specification file. The probe specification file contains the production rules of the constructs for which coverage statistics are to be generated, and the specification of the probes.

Driver instruments the grammar specification, depending upon the values in the tables constructed by the Specification parser.

3.2 Architecture of the Specification Parser

The specification parser consists of two modules:

Probe Specification Reader reads the probe specification file, and extracts the values of the different directives and production symbols. If the specification is syntactically correct, it provides these information to Spec driver.

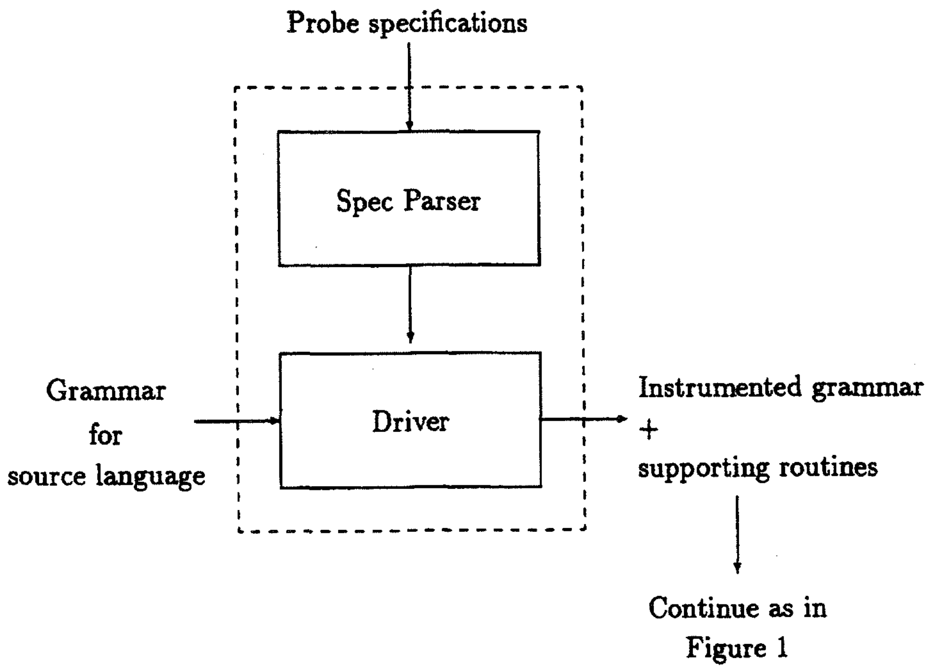

Probe specifications
Spec Parser
Driver
Grammar for source language
Instrumented grammar + supporting routines
Continue as in Figure 1

Figure 3: Functional Architecture of Generic TCA

Spec Driver constructs tables and probe databases from the probe specifications. One of the tables is the "production table", which stores the productions to be instrumented. This table stores the rule number, left hand side, and right hand side symbols of each of the production rule.

The probe specification may contain three types of probes, namely,

- probes to be put into the instrumented grammar specification,
- probes to be put in the instrumented source program, and
- directives to the driver to generate special probes

Spec driver differentiates between these probes through specific GTCA directives, and stores them in separate files. These files are later used by the driver module of the 'Grammar Instrumenter' to instrument the grammar specification, and for generating the supporting modules.

The functional architecture of Specification Parser is shown in Figure 4.

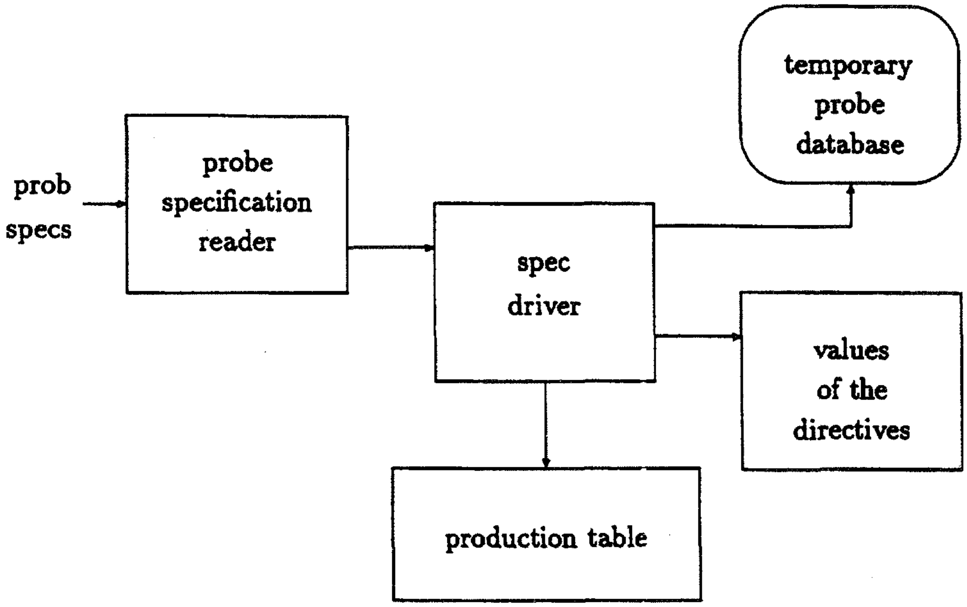


Figure 4. Functional Architectuıe of Spec Parser

3.3 Architecture of the Driver

The driver module of the *grammar Instrumenter* consists of four components:

Grammar specification reader: It reads the grammar specification file, and provides each production rule to the rule comparator. The grammar specification file also contains information like token declaration, variable declarations etc., apart from the productions. This information is provided to the instrumenter, in order to dump them as-it-is(or, with a little bit of customization) in the instrumented grammar specification file.

The production rules are supplied in the form of two components, viz., left hand side (lhs), and right hand side (rhs) of the rule, to the rule comparator. The lhs is a character string and rhs is a linked list of symbols. If the rule comparator cannot find any match, the production rule is dumped as-it-is in the instrumented grammar file. This is done by the instrumenter.

Rule comparator: It compares each production rule in the grammar specification with the rules in the probe specification files. The productions in the probe specification file have already been stored in the production table. The rules in the grammar specification are processed one rule at a time. If a match is found, rule comparator provides the rule number of the production to the probe generator. While comparing the rules, it discards the additional symbols, used in the probe specification for indicating the places of the probes in a production rule.

Probe generator: It generates the action part of the productions for which match has been found in the probe specification file. It uses the values of the directives, assigned by the user, for generating the action. If no GTCA directive is used to specify the probes in the probe specification, they are treated as the probes to be inserted into the instrumented source program. If the production in the grammar specification file already contains some actions, that is merged with the generated actions.

Instrumenter: It inserts the actions for the productions in the grammar file, for which a match is found in the probe specification file, and generates the instrumented grammar specification file. For accomplishing this task, it reads probe database, and uses the information provided by specification reader, rule comparator, and the probe generator. It also generates, the supporting routines, including the makefile and complete source distribution for the generated Test Coverage Analyzer.

The functional diagram of the Driver is shown in Figure 5.

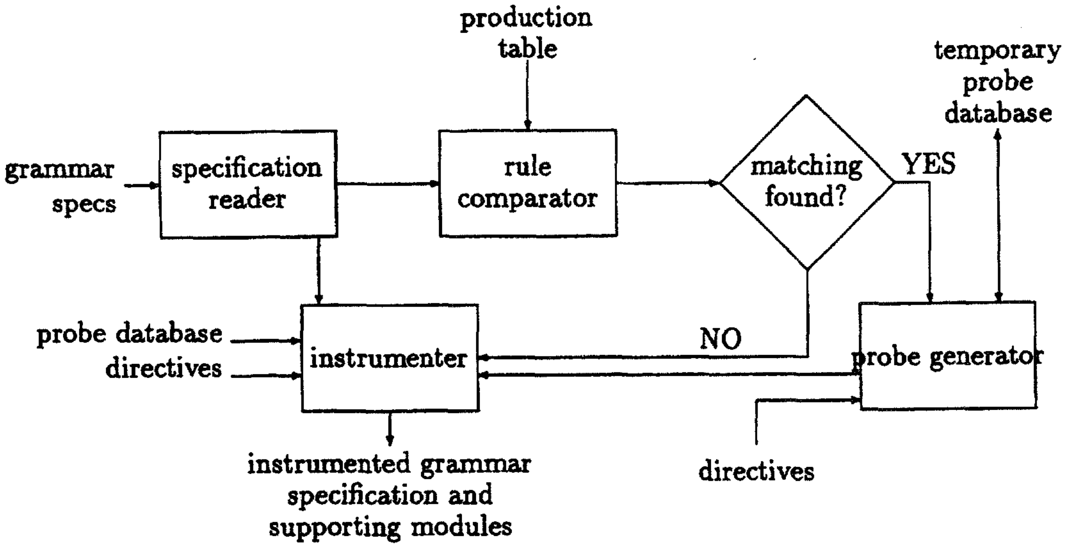


Figure 5. Functional Architectureof Driver

3.4 Output of GTCA

The output of GTCA comprises of the instrumented grammar file, and the complete source distribution for the language for which the Test Coverage Analyzer is to be developed. GTCA also generates the makefile for this distribution.

4 Experiences

The effort involved in developing Test Coverage Analyzers was compared by hand coding test coverage analyzer for C and Ada95 generating from GTCA. The quality of hand coded TCA and generated TCA was same.

The experimental results show that, using GTCA, one can develop reliable Test Coverage Analyzers at a rate much faster than(about 25 to 35 times) hand coded development. The probe specification contains minimal information, and are easy to specify. The specification language (similar to context free grammars) is flexible enough to incorporate generated and user defined probes effectively.

TCA (hand coded)	No. of lines (code)	Development Period (code)	No. of lines specs for GTCA	Development Period (specs)
C	5400	60 days	175	2 days
Ada95	4251	60 days	165	2 days

5 Summary

Development of a Test Coverage Analyzer is costly in terms of time and effort. However, TCAs are very useful in a software testing. This paper describes a generative approach for Test Coverage Analyzers. The implementation results have been found to be quite promising. The productivity gain is as much as 25-35 times of the hand coded development.

References

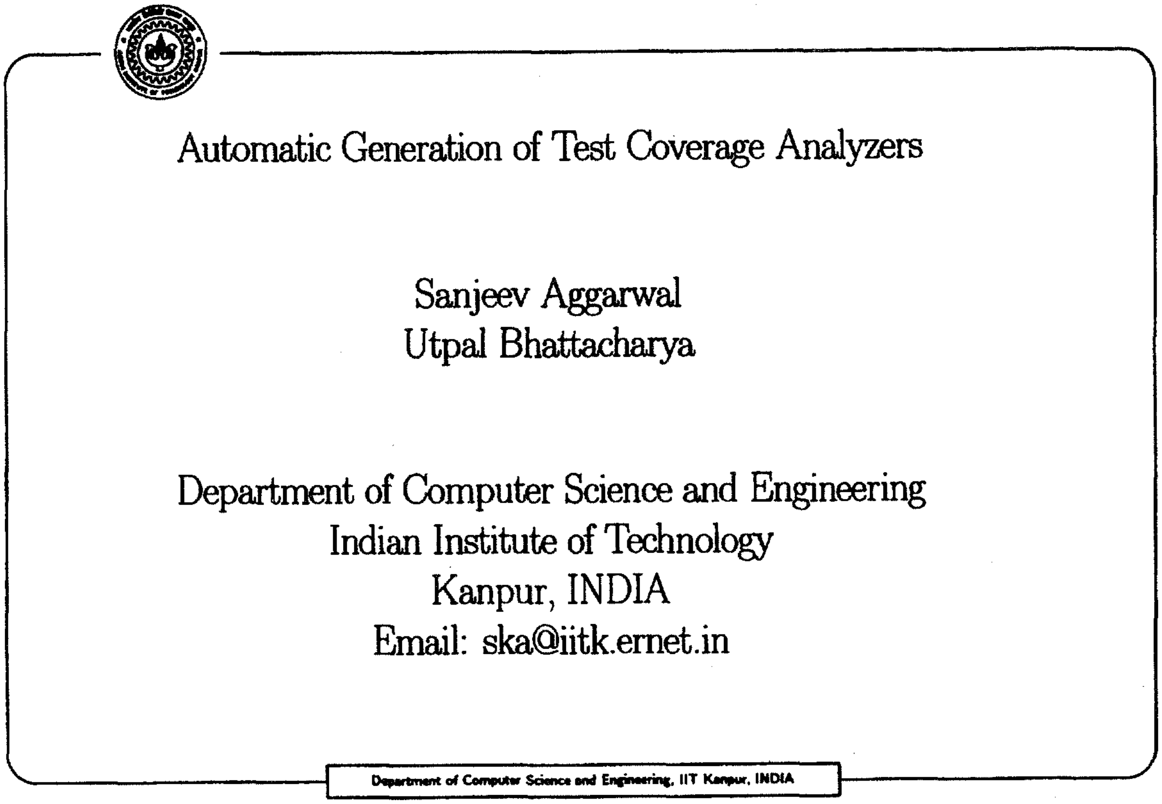
[Aga94] Hiralal Agarwal. Dominators, superblocks and program coverage. In *21st ACM SIGPLAN-SIGACT, Symposium on Principles of Programming languages*, pages 26–37, January 1994.

[Bei90] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, second edition, 1990.

[Cor96] Steve Cornett. Software test coverage analysis. Bullseye testing Technology, 1996. URL http://www.bullseye.com/coverage.html.

[Pro82] Robert L. Probert. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering*, SE-8(1):34–42, January 1982.

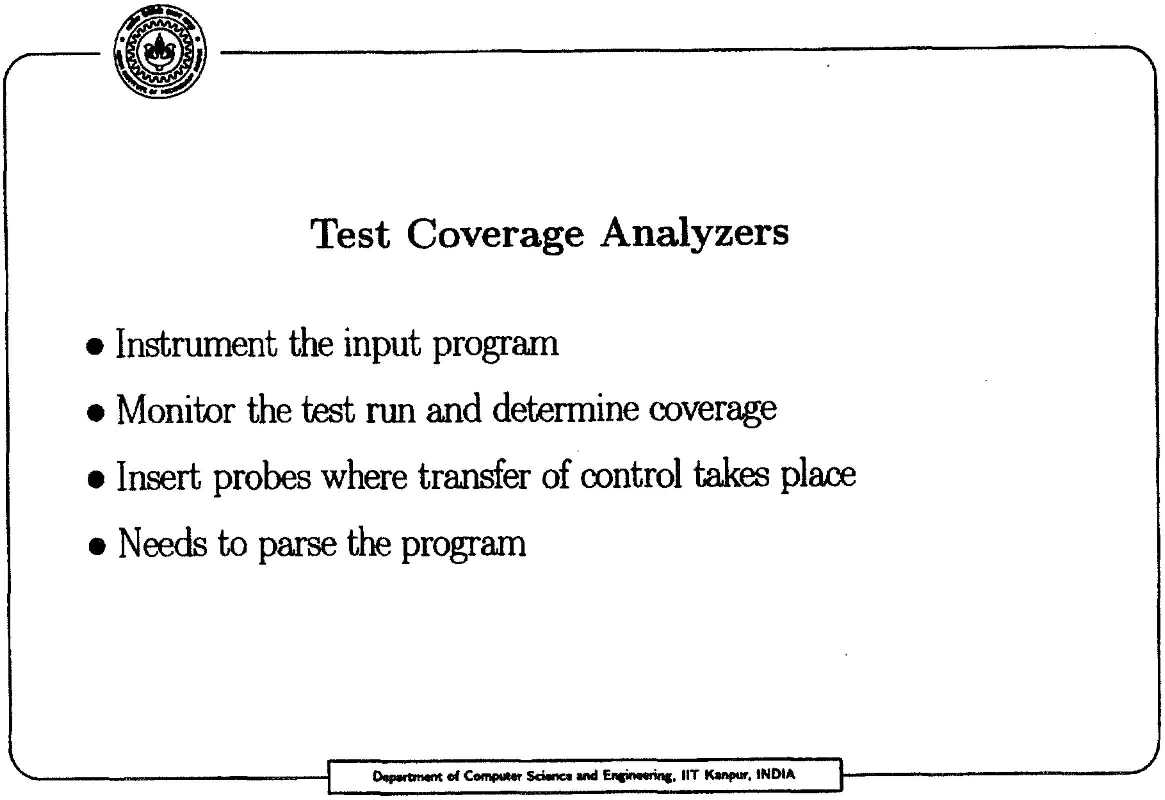
[RKC75] C. V. Ramamoorthy, K.H. Kim, and W. T. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE Transactions on Software Engineering*, SE-1(4):403–411, December 1975.



Automatic Generation of Test Coverage Analyzers

Sanjeev Aggarwal
Utpal Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur, INDIA
Email: ska@iitk.ernet.in



Test Coverage Analyzers

- Instrument the input program
- Monitor the test run and determine coverage
- Insert probes where transfer of control takes place
- Needs to parse the program

Functional Diagram

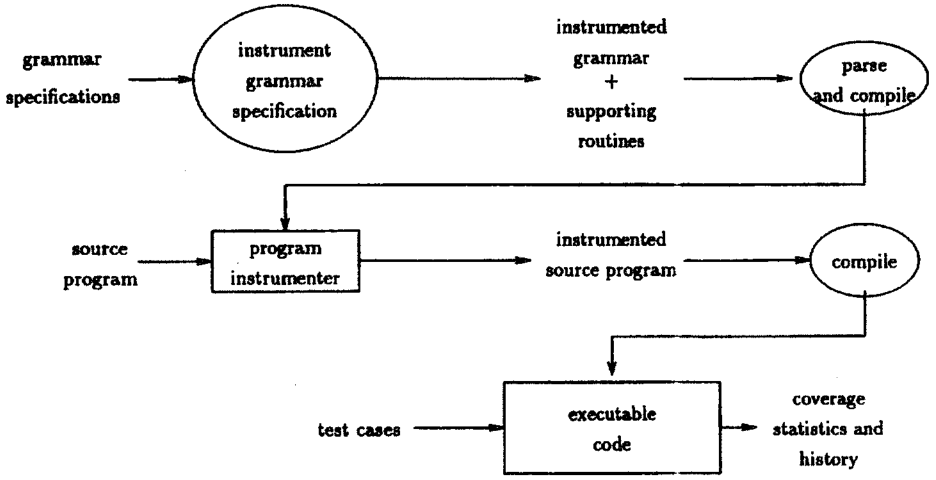
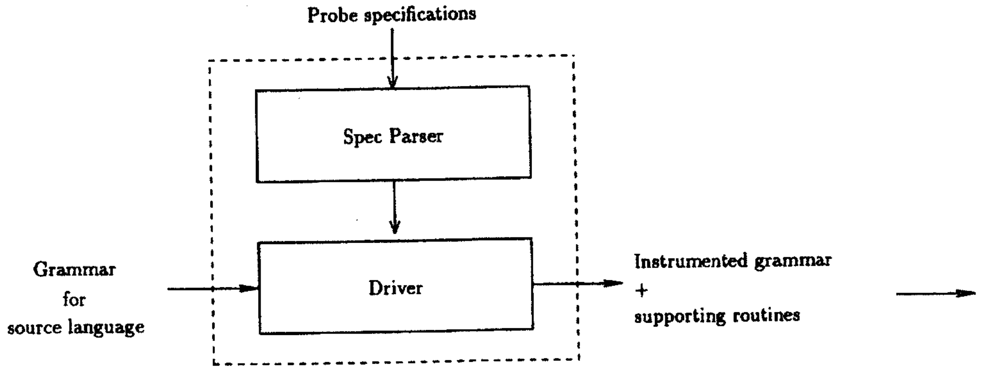

grammar specifications
instrument grammar specification
instrumented grammar + supporting routines
parse and compile
source program
program instrumenter
instrumented source program
compile
test cases
executable code
coverage statistics and history

Figure 1. Functional Diagram of Test Coverage Analyzer

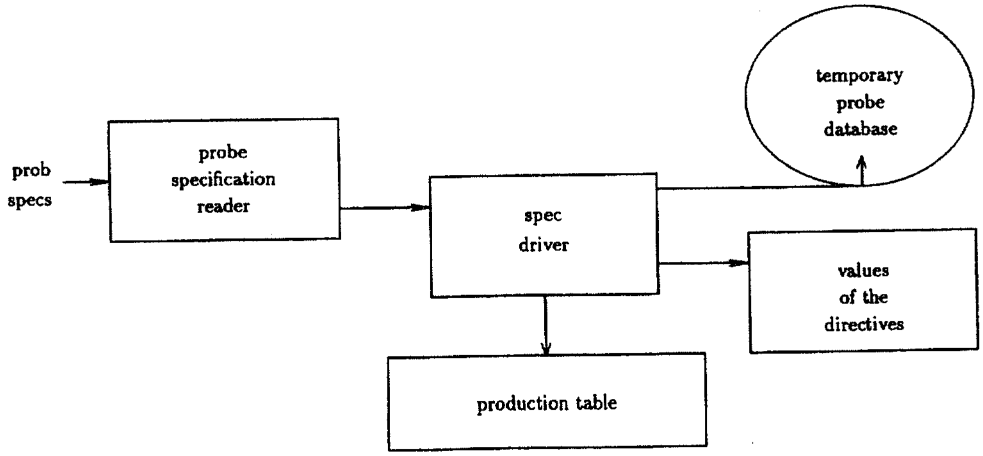
- Process of inserting probes is manual
- Most crucial phase
- Should be automated for fast development and reliability
- GENERATE Test Coverage Analyzers

Generate Test Coverage Analyzers

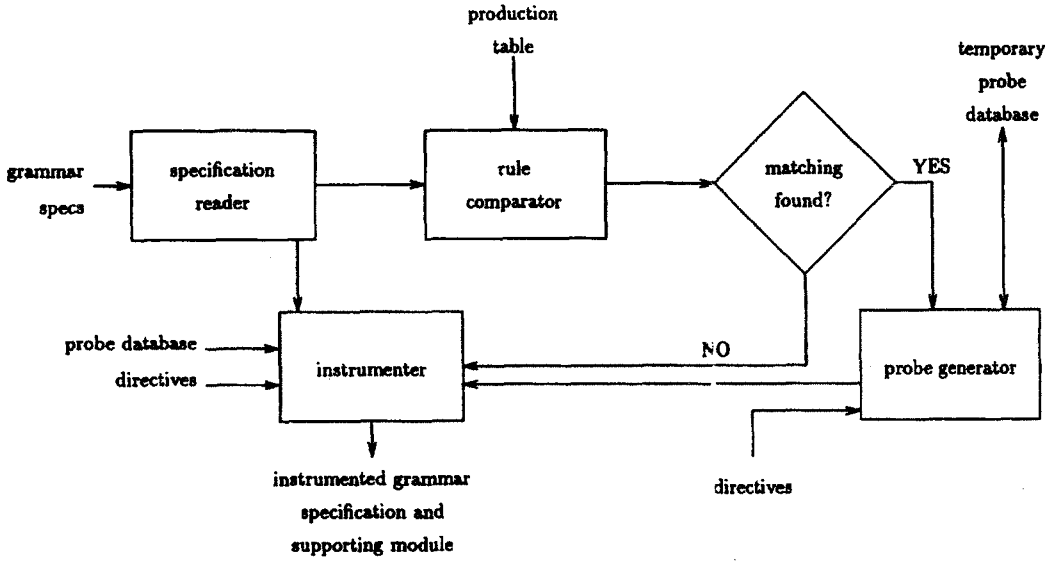
- Specify places where probes need to be inserted
- develop tools to process specification and insert probes



Specification Parser



Architecture of the Driver



production table

grammar specs

specification reader

rule comparator

matching found?

YES

NO

temporary probe database

probe database

directives

instrumenter

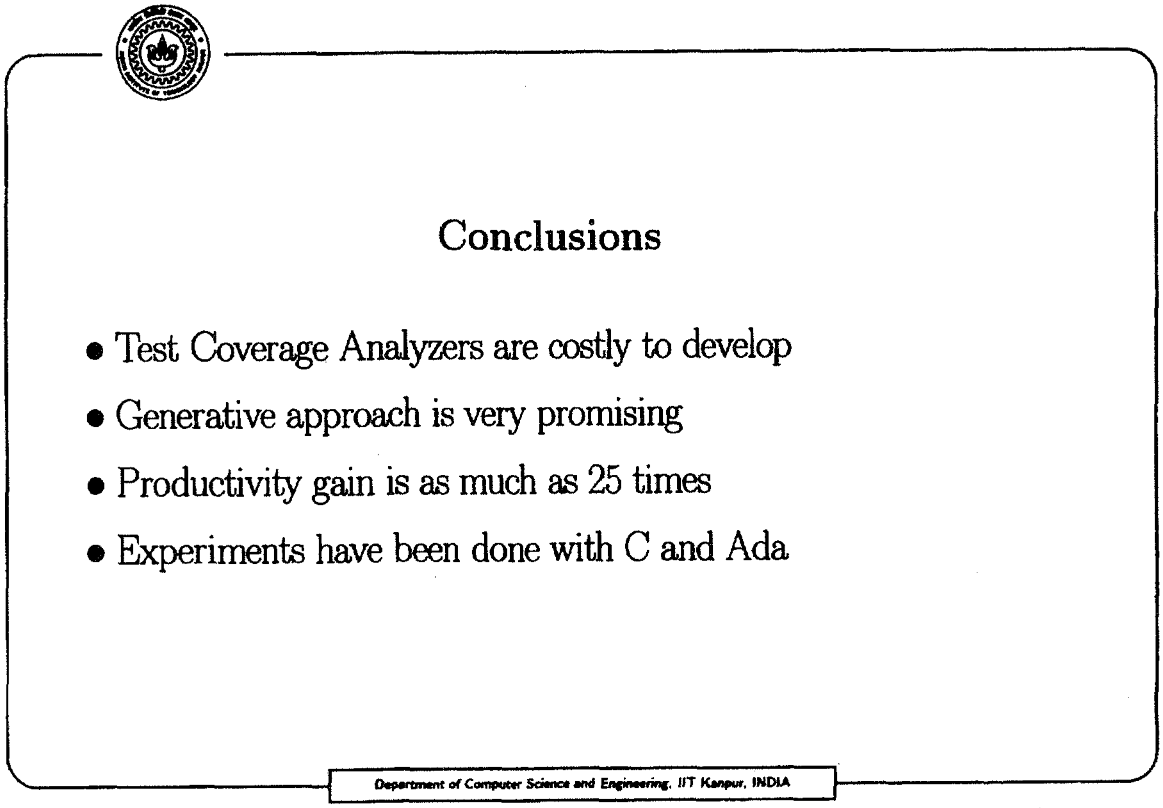
probe generator

directives

instrumented grammar specification and supporting module

Experience

TCA (hand coded)	No. of lines (code)	Development Period (code)	No. of lines specs for GTCA	Development Period (specs)
C	5400	60 days	175	2 days
Ada95	4251	60 days	165	2 days



Conclusions

- Test Coverage Analyzers are costly to develop
- Generative approach is very promising
- Productivity gain is as much as 25 times
- Experiments have been done with C and Ada

Department of Computer Science and Engineering, IIT Kanpur, INDIA

Page intentionally left blank

Page intentionally left blank

Measuring and Evaluating the Stability of Maintenance Processes

Norman F. Schneidewind
Code SM/Ss
Naval Postgraduate School
Monterey, CA 93943, U.S.A.

Voice: (408) 656-2719
Fax : (408) 656-3407
Email: schneidewind@nps.navy.mil

517-61

360870

p. 40

Abstract

In analyzing the stability of a maintenance process, it is important that it not be treated in isolation from the reliability and risk of deploying the software that result from applying the process. Furthermore, we need to consider the efficiency of the test effort that is a part of the process and a determinate of reliability and risk of deployment. Therefore, we were motivated to integrate these factors into a unified approach. Our contribution to maintenance is the integration and measurement of these factors so that the influence of maintenance actions and test effort on the reliability of the software and the risk of deploying it can be assessed. We use a safety critical application of National visibility -- the NASA Space Shuttle -- as an example application of the unified approach.

INTRODUCTION

Our purpose is to define and demonstrate, with a safety critical application of National visibility -- the NASA Space Shuttle -- the relationships among the following: 1) maintenance actions, 2) reliability, 3) test effort, and 4) risk to the safety of mission and crew of deploying the software after maintenance actions. These four factors are represented by the following types of metrics:

Maintenance actions: KLOC Change to the Code,

Reliability: Various reliability metrics (e.g., Total Failures, Remaining Failures, Time to Next Failure),

Test effort: Total Test Time, and

Risk: Remaining Failures and Time to Next Failure risk metrics.

We want to gain insight about the interaction of the maintenance process with software attributes like reliability and we apply these metrics for this purpose. When trends in these metrics over time are favorable (e.g., monotonic increasing reliability, monotonic decreasing test effort), we say the maintenance process is *stable* with respect to the software attribute (reliability, test effort). Conversely, when the trends are unfavorable (e.g., monotonic decreasing reliability, monotonic increasing test effort), we say the process is unstable. Thus we can formally define this concept as follows:

1. Definition of Maintenance Process *Stability*: If it is desirable that software attribute A increase (decrease) with time, then maintenance process M is *stable* with respect to A if A monotonically increases (decreases) with time t where t is time of release or test time, depending on the nature of A.

2. Definition of Maintenance Process *Instability*: If it is desirable that software attribute A increase (decrease) with time, then maintenance process M is *unstable* with respect to A if A monotonically decreases (increases) with time t where t is time of release or test time, depending on the nature of A.

When neither 1 nor 2 holds, we say that it is inconclusive as to whether M is stable or unstable. As a practical matter, if we are unable to conclude that a process is stable, we may at least be able to conclude that it is *not* unstable.

The use of the Shuttle application is appropriate to illustrate our purpose because it is essentially a large maintenance project that has been on-going from 1983 to the present time. Our contribution to maintenance is the integration and measurement of these four factors so that the influence of maintenance actions and test effort on the reliability of the software and the risk of deploying it can be assessed. The use of the Shuttle example is incidental to this purpose; it is a convenient case to use because of our experience with this application and the availability of data. We define, measure, and demonstrate both long term metrics -- those computed across a chronological sequence of releases -- and short term metrics -- those computed within a single release. The following relationships, using predicted and actual metrics, are discussed and illustrated:

LONG TERM RELATIONSHIPS AND METRICS

The following relationships are analyzed and metrics are computed over a sequence of releases:

1. Mean Time to failure (MTTF).

2. Total Failures normalized by KLOC CHANGE to the Code.

3. Total Maintenance Test Time normalized by KLOC CHANGE to the Code.

4. Remaining Failures normalized by KLOC Change to the Code

5. Time to Next Failure.

6. Remaining Failures Risk Metric.

7. Time to Next Failure Risk Metric.

SHORT TERM RELATIONSHIPS AND METRICS

The following relationships are analyzed and metrics are computed within a given release:

1. Total Maintenance Test Time versus Number of Remaining Failures.

2. Failure Rate versus Total Test Time.

The above relationships can be quantified. However we must also consider whether the functionality and complexity of the software has changed over time because these factors can have an effect on maintenance performance. There was no quantitative information available concerning increased functionality. We do know on a qualitative basis that functionality and complexity have been increasing over the life of the software. In a future research project we plan to use software complexity metrics in the maintenance process stability evaluation.

First we give brief descriptions of related work. Then we define the data and the application environment. This is followed by an analysis of relationships among maintenance, reliability, test effort, and risk, both long term (i.e., across releases)and short term (i.e., within a release). Lastly, conclusions are made concerning the feasibility of measuring and applying maintenance stability metrics.

RELATED RESEARCH AND PROJECTS

A number of useful related maintenance measurement and process projects have been reported in the literature. Briand, et al, developed a process to characterize software maintenance projects [BRI94]. They present a qualitative and inductive methodology for performing objective project characterizations to identify maintenance problems and needs. This methodology aids in determining causal links between maintenance problems and flaws in the maintenance organization and process.

Gefen and Schneberger, developed the hypothesis that maintenance proceeds in three distinct serial phases: corrective modification, similar to testing; improvement in function within the original specifications; and the addition of new applications that go beyond the original specifications [GEF96]. Their results from a single large information system, which they studied in great depth, suggested that software maintenance is a multi-period process. In the Shuttle maintenance process, in contrast, all three types of maintenance activities are performed concurrently and are accompanied by continuous testing.

Henry, et al, found a strong correlation between errors corrected per module and the impact of the software upgrade [HEN94].This information can be used to rank modules by their upgrade impact during code inspection in order to find and correct these errors before the software enters the expensive test phase.

Khoshgoftarr et al, used discriminant analysis in each iteration of their project to predict fault prone modules in the next iteration [KHO96]. This approach provided an advance indication of reliability and the risk of implementing the next iteration.

Pearse and Oman applied a maintenance metrics index to measure the maintainabilty of C source code before and after maintenance activities [PEA95]. This technique allowed the project engineers to track the "health" of the code as it was being maintained.

Pigoski and Nelson collected and analyzed metrics on size, trouble reports, change proposals, staffing, and trouble report and change proposal completion times [PIG94]. A major benefit of this project was the use of trends to identify the relationship between the productivity of the maintenance organization and staffing levels.

Sneed reengineered a client maintenance process to conform to the ANSI/IEEE Standard 1291, Standard for Software Maintenance [SNE96]. This project is a good example of how a standard can provide a basic framework for a process and can be tailored to the characteristics of the project environment.

Stark collected and analyzed metrics in the categories of customer satisfaction, cost, and schedule with the objective of focusing management's attention on improvement areas and tracking improvements over time [STA96]. This approach aided management in deciding whether to include changes in the current release, with possible schedule slippage, or include the changes in the next release.

Although there are similarities between these projects and our research, our work differs in that we integrate: 1) maintenance actions, 2) reliability, 3) test effort, and 4) risk to the safety of mission and crew of deploying the software after maintenance actions, for the purpose of analyzing and evaluating the stability of the maintenance process.

DATA AND EXAMPLE APPLICATION

We use software maintenance data from the NASA Space Shuttle, as shown in Table 1, which has two parts: 1 and 2. This table shows Operational Increments (Ois) of the Shuttle: OIA, ... ,OIQ, covering the period 1983-1997. An OI is defined as follows: a software system comprised of modules and configured from a series of builds to meet Shuttle mission functional requirements [SCH97]. For each of the OIs, we show the release date (the date of release by the contractor to NASA), post delivery total failures, failure severity, the maintenance change to the code in KLOC, and the time that was used to test the OI. Because the flight software is run continuously, around the clock, in simulation, test, or flight, total test time refers to continuous execution time from the time of release. For those seven OIs where there was a sufficient sample size (i.e., total failure count) -- OIA, OIB, OIC, OID, OIE, OIJ, and OIO -- to predict software reliability, we show launch date, mission duration, and reliability prediction date. Fortunately for the safety of the crew and mission, there have been few post delivery failures. Unfortunately, from the standpoint of prediction, there is a sparse set of observed failures from which to estimate reliability model parameters. Nevertheless, predictions were made prior to launch date for OIs with as few as five failures spanning many months of maintenance and testing. Lastly, three derived quantities are shown: MTTF and Total Failures/KLOC, where there are at least five failures, and Total Test Time/KLOC.

RELATIONSHIPS AMONG MAINTENANCE, RELIABILITY, TEST EFFORT, AND RISK

LONG TERM

It would be desirable for the maintenance effort to result in increasing reliability of software over a sequence of releases. A graph of this relationship over calendar time would show maintenance management whether the long term maintenance effort has been successful as it relates to reliability. In order to measure whether this is the case, we use both predicted and actual values of metrics. Predictions are necessary because we want to have an estimate of reliability in advance of deploying the software. If the predictions are favorable, they provide confidence that it is safe (i.e., acceptable risk) to deploy the software. If the predictions are unfavorable, we may decide to delay deployment and perform additional inspection and testing. Another reason for making predictions is to assess whether the maintenance process is effective in improving reliability and to do it sufficiently early during maintenance to improve the maintenance process. Actual values show in retrospect whether maintenance actions were successful in increasing reliability. Also, we do not want the test effort to be disproportionate to the amount of code that is changed and to the reliability that is achieved as a result of maintenance actions.

Measurement Criteria and Example Results

Mean Time to Failure

In the long term, we want Mean Time to Failure (MTTF) of an OI to **increase** over a sequence of OI releases, indicating **increasing** reliability. Ideally, it should increase monotonically. Practically, we would

Table 1-Part 1: Characteristics of Maintained Software Across *Shuttle* Releases

Operational Increment	Release Date	Launch Date	Mission Duration (Days)	Reliability Prediction Date	Total Post Delivery Failures	Failure Severity
A	9/1/83	No Flights		12/9/85	6	One 2 Five 3
B	12/12/83	8/30/84	6	8/14/84	10	Two 2 Eight 3
C	6/8/84	4/12/85	7	1/17/85	10	Two 2 Seven 3 One 4
D	10/5/84	11/26/85	7	10/22/85	12	Five 2 Seven 3
E	2/15/85	1/12/86	6	5/11/89	5	One 2N Four 3
F	12/17/85				2	Two 3
G	6/5/87				3	One 1 Two 3
H	10/13/88				3	Two 1N One 3
I	6/29/89				3	Three 3
J	6/18/90	8/2/91	9	7/19/91	7	Seven 3
K	5/2/91				1	One 1
L	6/15/92				3	One 1N One 2 One 3
M	7/15/93				1	One 3
N	7/13/94				1	One 3
O	10/18/95	11/19/96	18	9/26/96	5	One 2 Four 3
P	7/16/96				3	One 2 Two 3
Q	3/5/97				1	One 3

Table 1-Part 2: Characteristics of Maintained Software Across *Shuttle* Releases

Operational Increment	KLOC Change	Total Test Time (Days)	MTTF (Days)	Total Failures/ KLOC	Total Test Time/ KLOC
A	8.0	1078	179.7	0.750	134.8
B	11.4	4096	409.6	0.877	359.3
C	5.9	4060	406.0	1.695	688.1
D	12.2	2307	192.3	0.984	189.1
E	8.8	1873	374.6	0.568	212.8
F	6.6	412	206.0	0.300	62.4
G	6.3	3077	1025.7	0.476	488.4
H	7.0	540	180.0	0.429	77.1
I	12.1	2632	877.3	0.248	217.5
J	29.4	515	73.6	0.238	17.5
K	21.3	182			8.5
L	34.4	1337	445.7	0.087	38.9
M	24.0	386			16.1
N	10.4	121			11.6
O	15.3	344	68.8	0.327	22.5
P	7.3	272	90.0	0.411	37.3
Q	11.0	75			6.8

Notes:

a. Launch Date and Mission Duration are shown only where a reliability prediction was made. Predictions were not made for OIs where the sample size (i.e., *Total Failures*) was less than five.

b. Failure Count refers to post delivery failures.

c. **Severity Codes**:
1 : Severe Vehicle or Crew Performance Implications.
1N: Potentially Severity 1 but precluded by established operational procedures.
2 : Affects Ability to Complete Mission (Not a safety issue).
2N: Potentially Severity 2 but precluded by established operational procedures.
3 : Workaround Available, Minimal Effect on Procedures.
4 : Insignificant (Paperwork, etc.).

d. There were no pre-launch failures on OIE. Therefore there was no data for estimating model parameters. Prediction was made after launch.

look for an increasing trend.

Mean Time to Failure=Total Test Execution Time/Total Number of Failures During Test (1)

Total Failures

Similarly, we want Total Failures(and faults), normalized by KLOC Change in Code, to **decrease** over a sequence of OI releases, indicating that reliability is **increasing** with respect to code changes. Ideally, it should decrease monotonically. Practically, we would look for a decreasing trend.

Total Failures/KLOC=Total Number of Failures During Test/KLOC Change in Code on the OI (2)

Equations (1) and (2) are plotted in Figure 1 and Figure 2, respectively, against Release Time of OI. This is the number of months since the release of the OI, using "0" as the release time of OIA. The OIs are identified at the bottom of the plots. Both of these plots use actual values. Equation (1) is computed by dividing Total Test Time by Total Failures in Table 1. Equation (2) is computed by dividing Total Failures by KLOC Change in Table 1. Figures 1 and 2 do not provide consistent evidence that there is a long term increase in reliability. These plots would be used by management to assess whether there is long term stability in the maintenance process (i.e., whether reliability increases monotonically as changes are made to the code).

Total Maintenance Test Time

In the long term, we want the Total Maintenance Test Time, normalized by KLOC Change in Code, to **decrease** over a sequence of OI releases, indicating that test effort is **decreasing** with respect to code changes. Ideally, it should decrease monotonically. Practically, we would look for a decreasing trend.

Total Maintenance Test Time/KLOC=Total Test Time/KLOC Change in Code on the OI (3)

Equation (3) is plotted in Figure 3 against Release Time of OI. This plot uses actual values. Equation (3) is computed by dividing Total Test Time by KLOC Change in Code in Table 1. Figure 3 does not provide consistent evidence that there is a long term decrease in test effort. This plot would be used by management to assess whether testing is efficient with respect to the amount of code that has been changed.

Total Failures

Up to this point we have used only actual data fromTable 1 in the analysis. At this point we modify the analysis to use both predictions and actual data but only for seven OIs where we could make predictions. We develop additional tables for this purpose. Using the Schneidewind Model and the SMERFS software reliability tool [FAR93], we present prediction equations and make predictions for OIA, OIB, OIC, OID, OIE, OIJ, and OIO. We do not derive these equations because this has been done elsewhere [AIA93, SCH93, SCH92].

We predict Total Failures in the range $[1,\infty]$ (i.e., failures over the life of the software):

$F(\infty)=\alpha/\beta+X_{s-1}$ (4)

where the terms are defined as follows:

Figure 1. Mean Time To Failure Across Releases

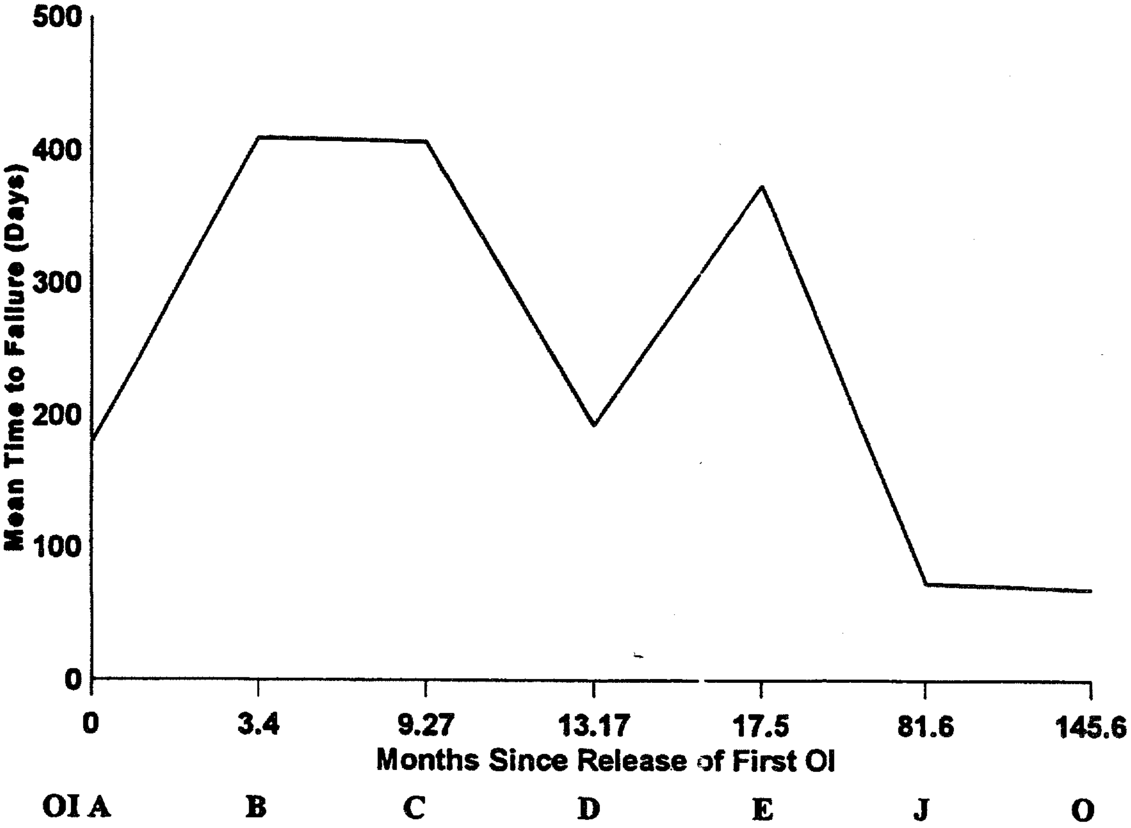


Figure 2. Failures per KLOC Across Releases

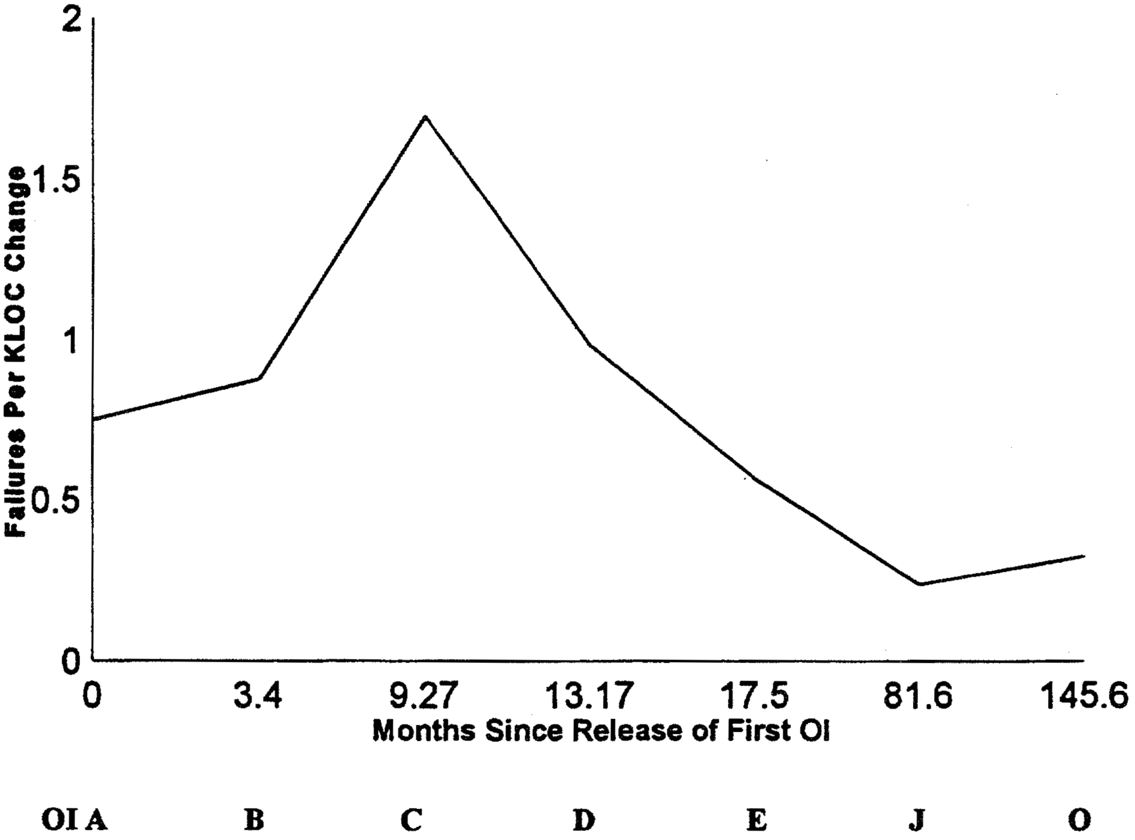
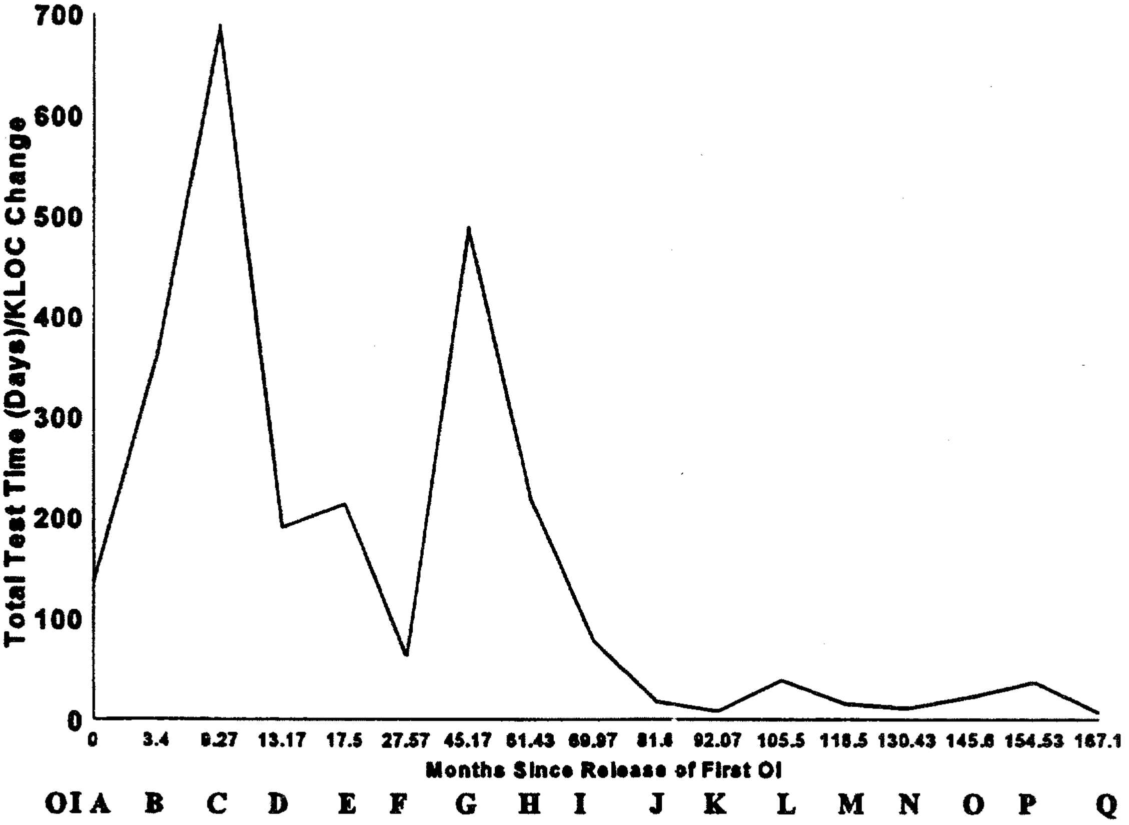


Figure 3. Test Time per KLOC Across Releases



s: starting interval for using observed failure data in parameter estimation,
α: failure rate at the beginning of interval s,
β: negative of derivative of failure rate divided by failure rate (i.e., relative failure rate), and
X_{s-1}:observed failure count in the range [1,s-1].

We never know the actual number of Total Failures because additional failures could occur in the future, particularly in a system like the Shuttle where current OIs reuse much of the code that was developed many years ago in earlier OIs. Therefore, we use the approximation that Total Failures equals the number of failures observed over a sufficiently long period of test time (i.e., years in the case of the Shuttle). This approximation tends to underestimate Total Failures.

Predicted and actual Total Failures are shown in Table 2 and normalized by KLOC Change in Code in Table 3. The actual values in Table 2 are repeated from Table 1 for the selected OIs. As in equation (2), we want equation (4) and actual Total Failures, normalized by KLOC Change in Code, to **decrease** over a sequence of releases. The data in Table 3 are inconclusive about a long term increase in reliability.

Predicted values would be used as an alert to management that the number of failures anticipated over the life of the software is either acceptable or unacceptable; if the latter, it may be necessary to improve the maintained product or the maintenance process. Actual values would be used retrospectively to measure the reliability of the software resulting from maintenance actions.

Remaining Failures

To obtain predicted remaining failures r(t) at time t, we use equation (5) [AIA93, KEL95, SCH93]:

$$r(t)=(\alpha/\beta)-X_{s,t}=F(\infty)-X_t \qquad (5)$$

where $X_{s,t}$ is observed failure count in the range [s,t] and X_t observed failure count in the range [1,t].

As in the case of Total Failures, we never know the actual number of Remaining Failures. Therefore we use the approximation that at any time t, Remaining Failures is the difference between actual number of Total Failures and X_t. This approximation tends to underestimate Remaining Failures.

Predicted and actual Remaining Failures are shown in Table 2 and normalized by KLOC Change in Code in Table 3. Again, we want equation (5) and actual Remaining Failures, normalized by KLOC Change in Code, to **decrease** over a sequence of releases. Predicted and actual values are plotted for seven OIs in Figure 4. The two plots have similar shapes, but they are inconclusive about a long term increase in reliability.

Predicted values would be used as an alert to management of the number of residual faults in the code and the failures that may occur as a consequence. The risk to safety may or may be not be acceptable to management. If the latter, it may be necessary to improve the maintained product or the maintenance process. Actual values would be used retrospectively to measure the reliability of the software and the risk of deploying it, resulting from maintenance actions.

Time to Next Failure

To predict the Time for the Next F_t Failures to occur, when the current time is t, we use equation (6)

Table 2: Reliability of Maintained Software: Predictions versus Actuals

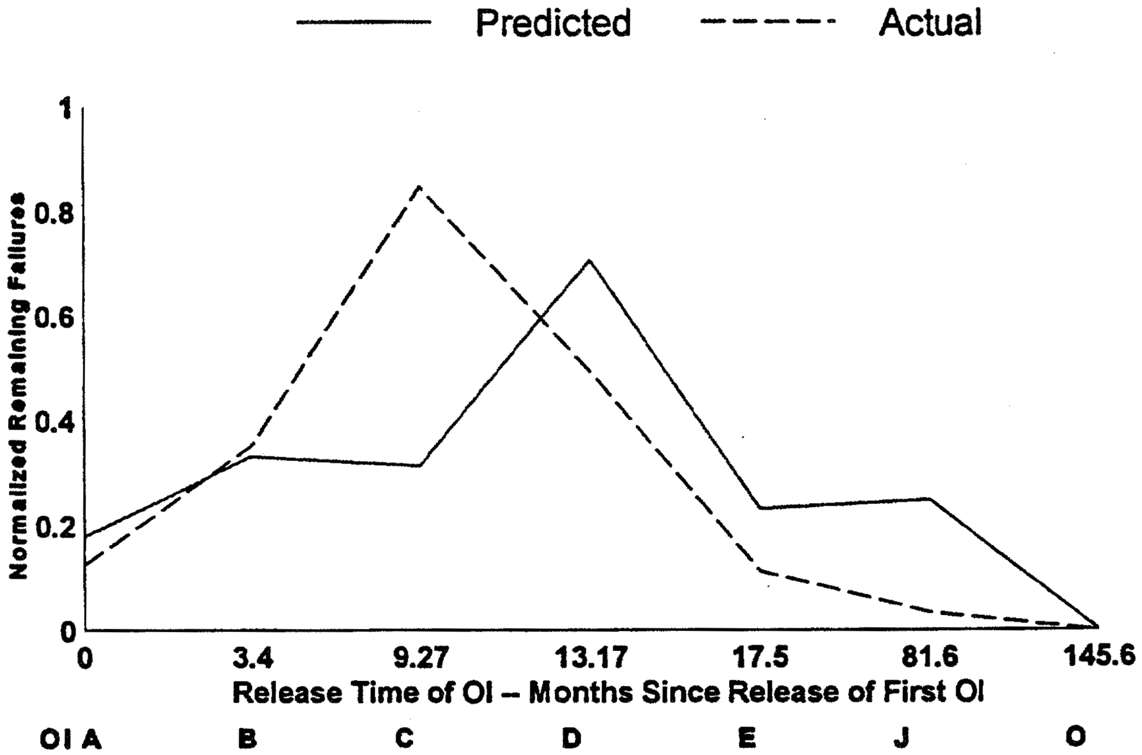
	Total Failures		Remaining Failures		Time to Next Failure	
Operational Increment	Predicted	Actual	Predicted	Actual	Predicted (Intervals)	Actual (Intervals)
OIA	6.44	6	1.44	1	19.70	8.26
OIB	9.78	10	3.78	4	3.49	1.43
OIC	6.85	10	1.85	5	2.03	5.83
OID	14.60	12	8.60	6	6.54	4.77
OIE	6.04	5	2.04	1	19.27	10.90
OIJ	13.32	7	7.32	1	2.57	3.97
OIO	5.02	5	0	0	No Failure	No Failure

Note: Interval length is 30 days.

Table 3: Total & Remaining Failures Normalized by Maintenance Change to Code

		Normalized Total Failures		Normalized Remaining Failures	
Operational Increment	KLOC Change	Predicted	Actual	Predicted	Actual
OIA	8.0	0.805	0.750	0.180	0.125
OIB	11.4	0.857	0.877	0.332	0.351
OIC	5.9	1.161	1.695	0.314	0.847
OID	12.2	1.197	0.984	0.705	0.492
OIE	8.8	0.686	0.568	0.232	0.114
OIJ	29.4	0.453	0.238	0.249	0.034
OIO	15.3	0.328	0.327	0	0

Figure 4. Reliability of Maintained Software -- Remaining Failures Normalized by Change to Code



[AIA93, SCH93]:

$$T_F(t)=[(\log[\alpha/(\alpha-\beta(X_{s,t}+F_t)])/\beta]-(t-s+1)$$
$$\text{for } (\alpha/\beta)>(X_{s,t}+F_t) \qquad (6)$$

The terms in $T_F(t)$ have the following definitions:
t: Current interval;
$X_{s,t}$: Observed failure count in the range [s,t]; and
F_t: Given number of failures to occur after interval t.

The usual application of equation (6) is to predict the Time to Next Failure (i.e., $F_t=1$). This is the case in this analysis. Predicted and actual Time to Next Failure are shown in Table 2. We want equation (6) to **increase** over a sequence of releases. Predicted and actual values are plotted for six OIs in Figure 5. The two plots have similar shapes, but they are inconclusive concerning whether there is a long term increase in reliability.

Predicted values would be used as an alert to management of how long the software could continue to operate before the next failure occurs. The risk to safety may or may be not be acceptable to management. If the latter, it may be necessary to improve the maintained product or the maintenance process. Actual values would be used retrospectively to measure the reliability of the software and the risk of deploying it, resulting from maintenance actions.

Criteria for Safety

Now we apply the Remaining Failures and Time to Next Failure metrics to assess the risks to safety resulting from maintenance actions [SCH97]. As pointed out in [FAI94], a risk becomes a problem when the value of a quantitative metric crosses a predetermined threshold. Thus there are two parts of risk management: setting thresholds beyond which some corrective action is required and determining ahead of time what that corrective action will be. In the case of maintenance, we establish Remaining Failures and Time to Next Failure thresholds. If these are exceeded, our action would be to correct the product and the maintenance process that produced the product.

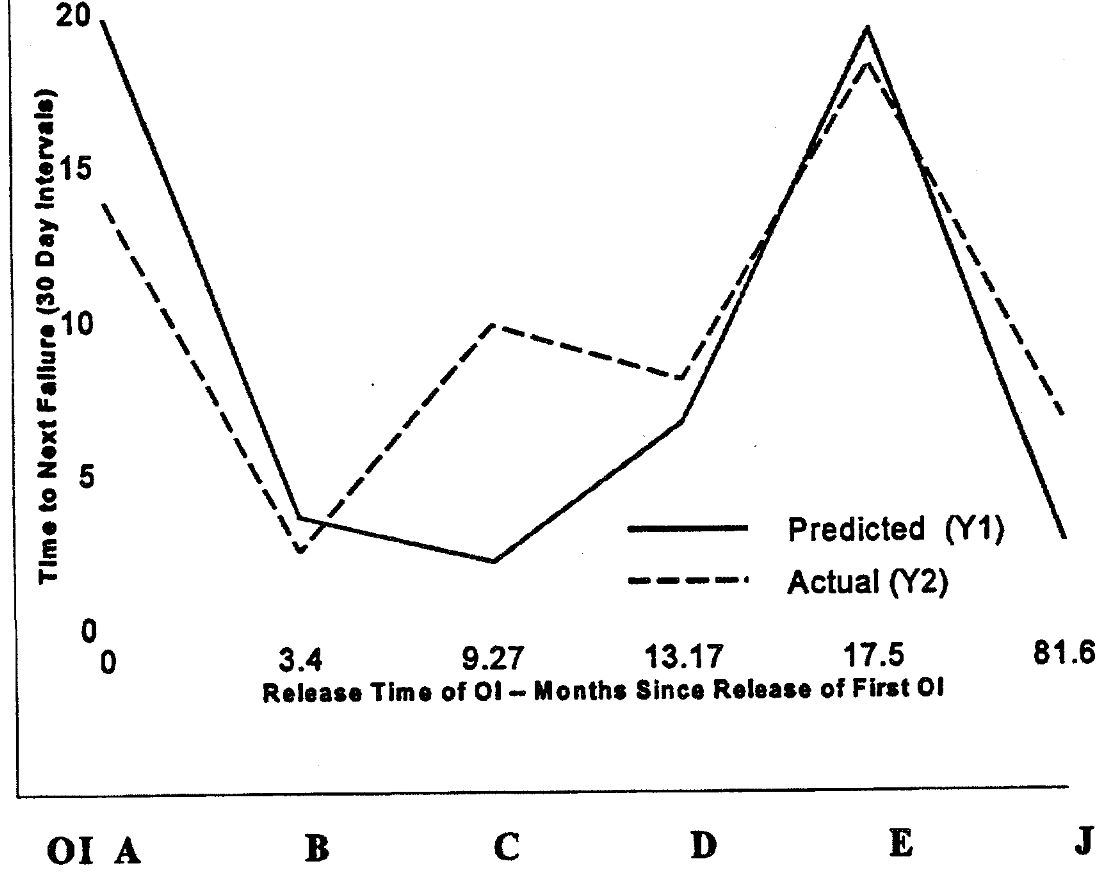
If we define our safety goal as the reduction of failures that would cause loss of life, loss of mission, or abort of mission to an acceptable level of risk, then for software to be ready to deploy, after having been tested for total time t_t, we must satisfy the following criteria:

1) predicted remaining failures $r(t_t)<r_c$, (7)
where r_c is a specified critical value , and

2) predicted time to next failure $T_F(t_t)>t_m$, (8)
where t_m is mission duration.

For systems that are tested and operated continuously like the Shuttle, t_t, $T_F(t_t)$, and t_m are measured in execution time. Note that, as with any methodology for assuring software safety, we cannot guarantee safety. Rather, with these criteria, we seek to reduce the risk of deploying the software to an acceptable level.

Figure 5. Reliability of Maintained Software -- Time to Next Failure



Remaining Failures Criterion

Using the assumption that the faults that cause 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. The reason for this choice is that one or more Remaining Failures would constitute unacceptable risk for safety critical systems. This is the threshold used by the Shuttle software managers. 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, $r(t_t)<1$ would mean that $r(t_t)$ must be less than one failure, independent of severity level.

If we predict $r(t_t) \geq r_c$, we would continue to test for a total time $t_t'>t_t$ that is predicted to achieve $r(t_t')<r_c$, using the assumption that we will experience more failures and correct more faults so that the Remaining Failures will be reduced by the quantity $r(t_t)-r(t_t')$. 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. We know from Dijkstra's dictum that we cannot demonstrate the absence of faults [DIJ70]; however we can reduce the risk of failures occurring to an acceptable level, as represented by r_c.

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 $T_F(t_t) \leq t_m$, we would continue to test for a total time $t_t''>t$ that is predicted to achieve $T_F(t_t'')>t_m$, using 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 $T_F(t_t'')-T_F(t_t)$. 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.

Risk Assessment

The amount of Total Test Time t_t can be considered a measure of the degree to which software reliability goals have been achieved. This is particularly the case for systems like the Shuttle where the software is subjected to continuous and rigorous testing for several years in multiple facilities, using a variety of operational and training scenarios (e.g., by Lockheed-Martin in Houston, by NASA in Houston for astronaut training, and by NASA at Cape Canaveral). We can view t_t as an input to a risk reduction process, $r(t_t)$ and $T_F(t_t)$ as the outputs, and ${}_cr$ and${}_mt$ as "risk criteria levels" of safety that control the process. While we recognize that Total Test Time is not the only consideration in developing test strategies and that there are other important factors, like the consequences for reliability and cost, in selecting test cases [WEY95], nevertheless, for the foregoing reasons, Total Test Time has been found to be strongly positively correlated with reliability growth for the Shuttle [SCH92].

Remaining Failures

We can formulate the mean value of the Risk Criterion Metric (RCM) for criterion 1) as follows:

$$\text{RCM } r(t_t)=(r(t_t)-r_c)/r_c=(r(t_t)/r_c)-1 \qquad (9)$$

Positive, zero, and negative values of equation (9) correspond to $r(t_t)>r_c$, $r(t_t)=r_c$, and $r(t_t)<r_c$, respectively. These are the critical, neutral, and desired regions, respectively. Predicted and actual values of equation (9) are shown in Table 4 for $r_c=1$. We want equation (9) to **decrease** (become more negative) over a sequence of releases. Predicted and actual values are plotted for seven OIs in Figure 6. The two plots have similar shapes, but they are inconclusive concerning whether there is a long term decrease in risk.

Predicted values would be used as an alert to management of the risk of deploying the software due to possible residual faults and failure occurrences. The risk to safety may or may be not be acceptable to management. If the latter, it may be necessary to improve the maintained product or the maintenance process. Actual values would be used retrospectively to measure the risk of deploying software, resulting from maintenance actions.

Time to Next Failure

Similarly, we can formulate the mean value of the Risk Criterion Metric (RCM) for criterion 2) as follows:

$$\text{RCM } T_F(t_t)=(t_m-T_F(t_t))/t_m=1-(T_F(t_t))/t_m \qquad (10)$$

Positive, zero, and negative values of equation (10) correspond to $T_F(t_t)<t_m$, $T_F(t_t)=t_m$, and $T_F(t_t)>t_m$, respectively. These are the critical , neutral, and desired regions, respectively. Predicted and actual values of equation (10) are shown in Table 4 for t_m = mission duration of the OI. We want equation (10) to **decrease** (become more negative) over a sequence of releases. The data in Table 4 are inconclusive concerning whether there is a long term decrease in risk.

Predicted values would be used as an alert to management of the risk of deploying the software due to possibility failures occurring during the mission. The risk to safety may or may be not be acceptable to management. If the latter, it may be necessary to improve the maintained product or the maintenance process. Actual values would be used retrospectively to measure the risk of deploying the software, resulting from maintenance actions.

Summary

The type of results shown in Tables 1, ... ,4 and Figures 1, ... ,6 would be an alert to management to investigate whether the inconsistency in results is caused by: 1) greater functionality and complexity in the software over a sequence of releases, 2) a maintenance process that needs to be improved, or 3) a combination of these causes. Although we cannot conclude that any of the above metrics demonstrate a stable maintenance process for the Shuttle, we *can* conclude that the maintenance process is *not* unstable (e.g., monotonically decreasing MTTF).

SHORT TERM

In addition to the long term maintenance stability criteria, described earlier, it would be desirable for the maintenance effort to result in increasing reliability of the software within each OI's test history. Also, we want the test effort to be efficient in finding residual faults, for a given OI.

Table 4: Reliability Risk Assessment of Maintenance Actions

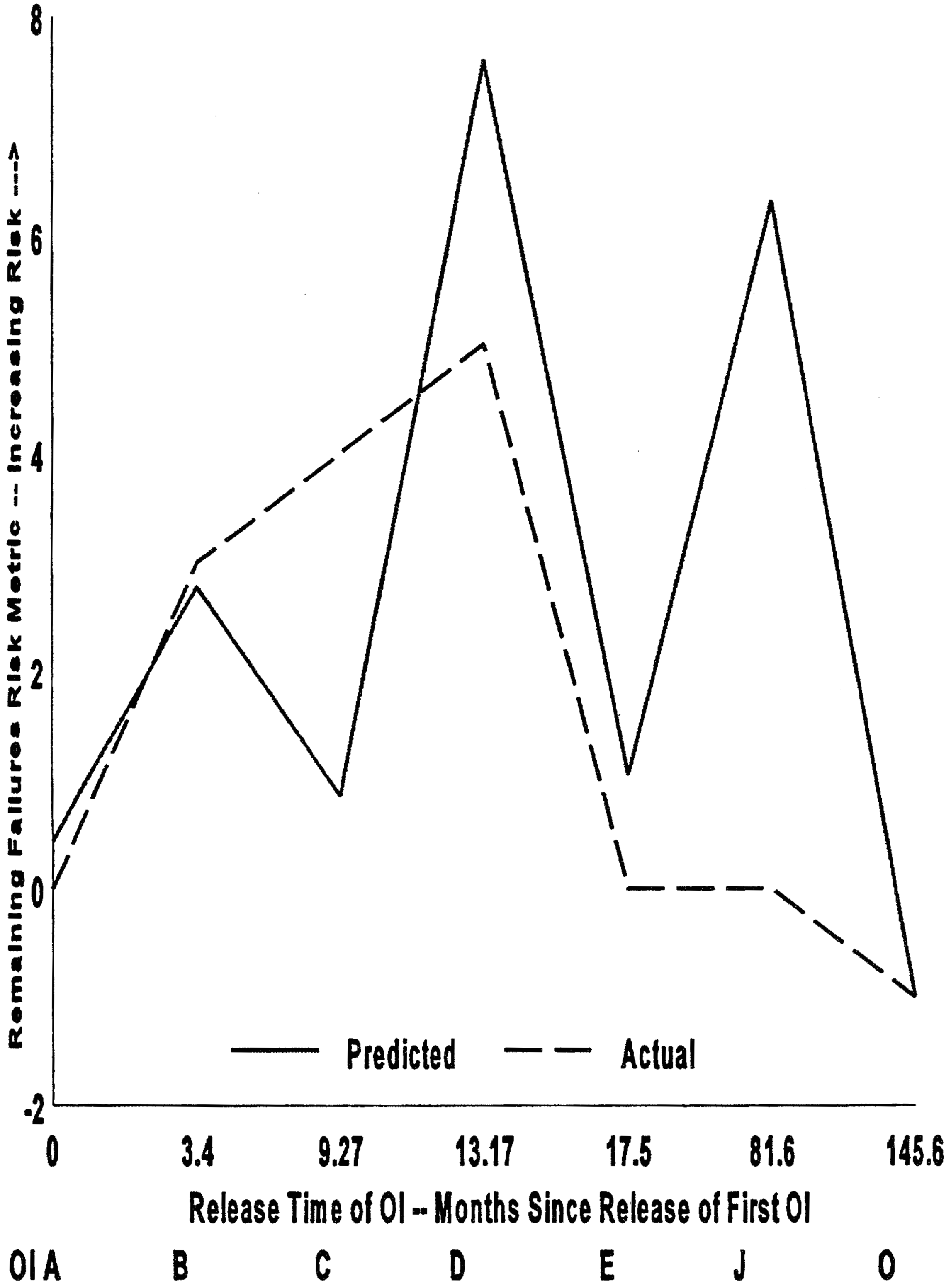
	Remaining Failures Risk Metric for One Remaining Failure		Time to Next Failure Risk Metric for Mission Duration		
Operational Increment	Predicted	Actual	Mission Duration (Days)	Predicted	Actual
OIA	0.44	0.00	No Flights		
OIB	2.78	3.00	6	-16.45	-6.30
OIC	0.85	4.00	7	-7.83	-24.35
OID	7.60	5.00	7	-27.43	-19.74
OIE	1.04	0.00	6	-95.35	-53.50
OIJ	6.32	0.00	9	-7.57	-12.23
OIO	-1.00	-1.00	18	No Failure	No Failure

Table 5: Total Maintenance Test Time: Predictions versus Actuals

R	Operational Increment Total Maintenance Test Time (30 Day Intervals) Required to Achieve "R" Remaining Failures: P=Predicted, A=Actual											
	OIA		OIB		OIC		OID		OIE		OIJ	
	P	A	P	A	P	A	P	A	P	A	P	A
1	34.2	27.7	17.0	87.5	40.2	43.8	58.1	65.0	72.3	52.0	48.8	13.2
2			11.6	10.4	31.6	27.1	43.6	58.3				
3			8.4	9.6	26.5	16.9	35.1	45.2				
4			6.2	8.2	22.9	13.9	29.0	23.7				
5					20.1	7.4	24.4	17.5				
6							20.5	12.7				

Note: A blank entry means there were no more remaining failures for the Operational Increment.

Figure 6. Remaining Failures Risk Metric for 1 Remaining Failure



Measurement Criteria and Example Results Using One OI

In the analysis that follows we use predictions and actual (observed) data for one OI -- OID.

Total Maintenance Test Time

The predicted Total Maintenance Test Time required to achieve a specified number of Remaining Failures, $r(t_t)$, at time t_t, is given by equation (11) [AIA93, SCH93]:

$$t_t=[\log[\alpha/(\beta[r(t_t)])]]/\beta+(s-1) \qquad (11)$$

Predicted and actual Total Maintenance Test Time are shown in Table 5 for six OIs. Equation (11) is plotted for OID in Figure 7 against given number of Remaining Failures. The two plots have similar shapes and show the typical asymptotic characteristic of reliability (i.e., Remaining Failures) versus Total Test Time: the plots indicate the possibility of big gains in reliability in the early part of testing; eventually the gains become marginal as testing continues. Predicted values would be used by management to gauge how much maintenance test effort would be required to achieve desired reliability goals and whether the predicted amount of test time is technically and economically feasible. Actual values would be used retrospectively to judge whether the maintenance test effort has been efficient in relation to the reliability that has been achieved.

Failure Rate

In the short term, we want Failure Rate (1/MTTF) of an OI to **decrease** over an OI's Total Test Time, indicating **increasing** reliability. Ideally, it should decrease monotonically. Practically, we would look for a decreasing trend, after an initial period of instability (i.e., increasing rate).

Failure Rate=Total Number of Failures During Test/Total Test Execution Time (12)

Equation (12) is plotted for OID in Figure 8 against Total Test Time since the release of OID. Equation (12) is computed from a listing of complete failure history of OID This listing is not shown because of its length. Figure 8 *does* show that short term stability is achieved (i.e., failure rate asymptotically approaches zero with increasing test time). These plots would be used by management to assess whether there is long term stability in the maintenance process (i.e., whether reliability increases as changes are made to the code).

Summary

The type of results shown in Table 5 and Figures 7 and 8 would indicate to management whether the maintenance process is stable in the short term. Instability (i.e., monotonically increasing failure rate over test time) would be an alert to management to investigate whether this is caused by: 1) greater functionality and complexity of the OI as it is being maintained, 2) a maintenance process that needs to be improved, or 3) a combination of these causes.

CONCLUSIONS

We conclude, based on both predictive and retrospective use of maintenance, reliability, test, and risk metrics, that it is feasible to measure the stability of a maintenance process and to integrate these factors

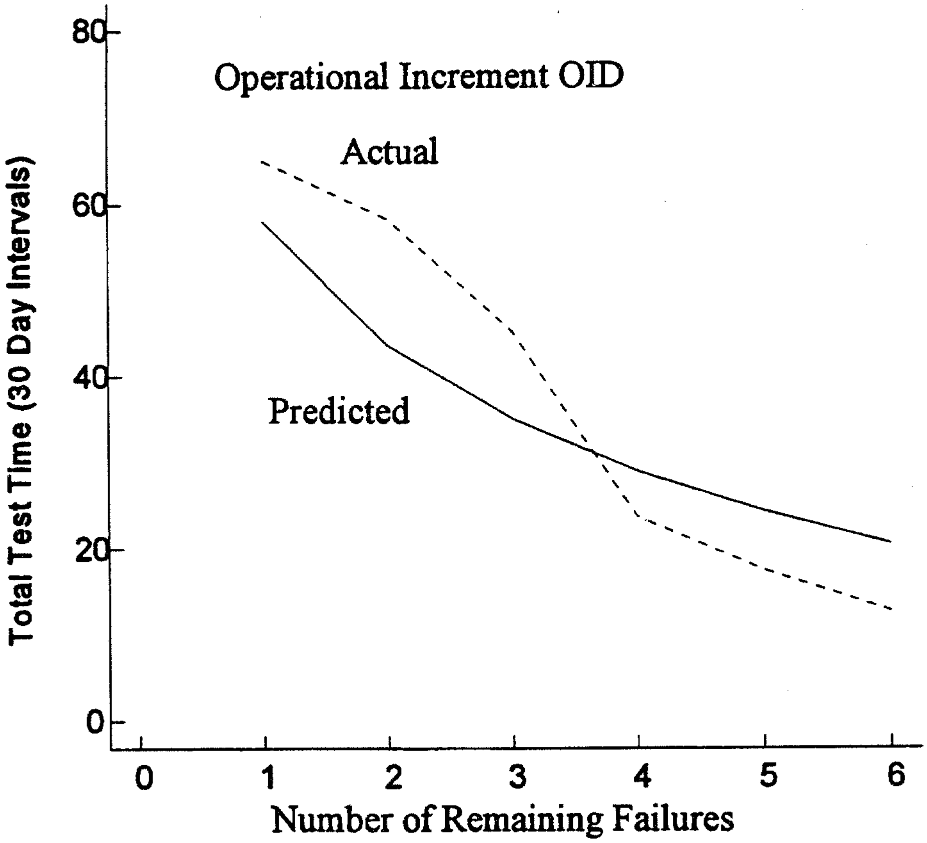


Figure 7. Total Maintenance Test Time to Achieve Remaining Failures

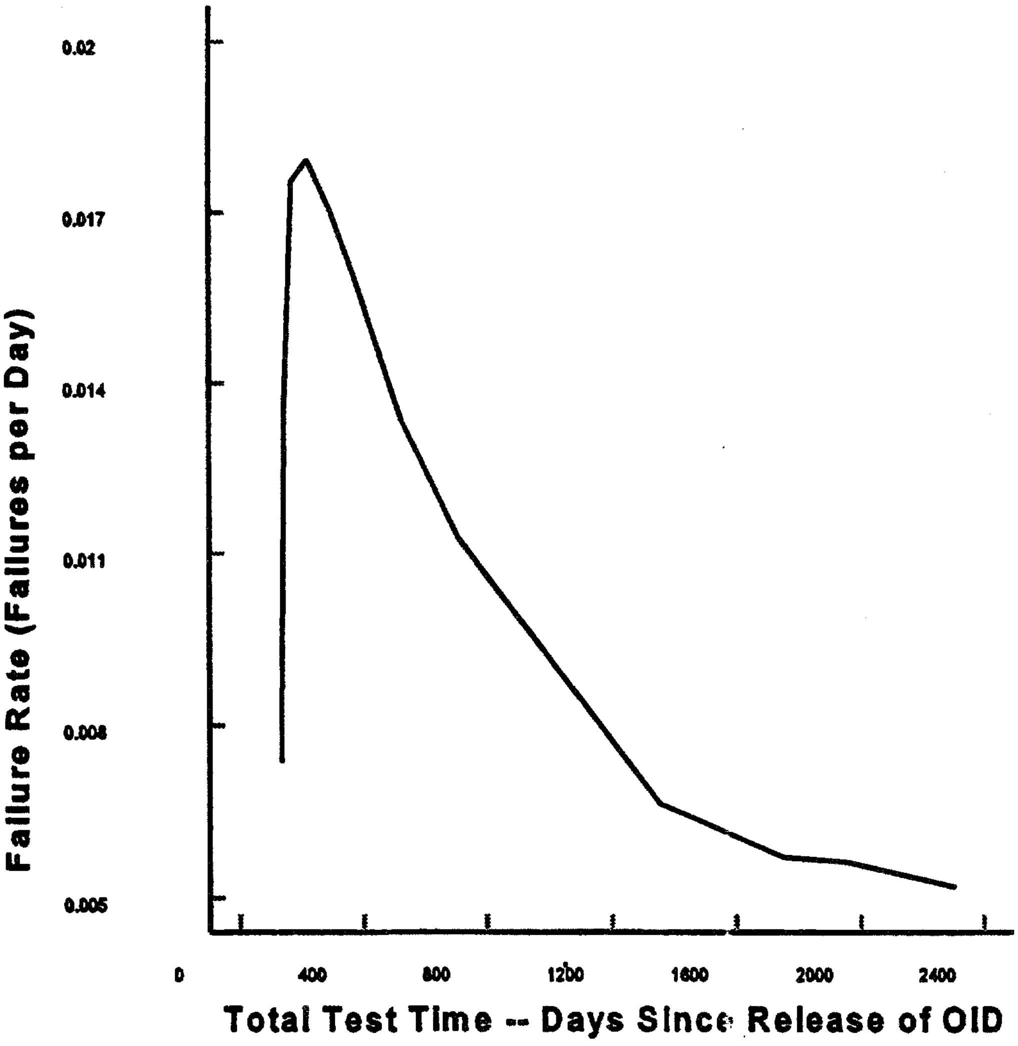


Figure 8. Reliability of Maintained Software: Failure Rate of OID

into a unified approach for assessing the impact of the maintenance process on the reliability and risk of deploying the software. Future research goals are to relate maintenance process to various software characteristic metrics and to use a larger sample of failure data by combining pre release and post release failure data.

Acknowledgments

We acknowledge the support provided for this project by Dr. William Farr, Naval Surface Warfare Center; Mr. Allen Nikora of the Jet Propulsion Laboratory; the NASA IV&V Center; U.S. Marine Corps Tactical Systems Support Activity; and Mr. Ted Keller, Ms Julie Barnard, and Ms. Patti Thornton of Lockheed-Martin.

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.

[BRI94] Lionel C. Briand, Victor R. Basili, and Yong-Mi Kim, "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.

[DIJ70] E. W. Dijkstra, "Structured Programming", Software Engineering Techniques, eds. J. N. Buxton and B. Randell, NATO Scientific Affairs Division, Brussels 39, Belgium, April 1970, pp. 84-88.

[FAI94] Richard Fairley, "Risk Management for Software Projects", IEEE Software, Vol. 11, No. 3, May 1994, pp. 57-67.

[FAR93] William H. Farr and Oliver D. Smith, Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide, NAVSWC TR-84-373, Revision 3, Naval Surface Weapons Center, Revised September 1993.

[GEF96] David Gefen and Scott L. Schneberger, The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 134-141.

[HEN94] Joel Henry, Sallie Henry, Dennis Kafura, and Lance Matheson, "Improving Software Maintenance at Martin Marietta", IEEE Software, Vol. 11, No.4, July 1994, pp. 67-75.

[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.

[KHO96] Taghi M. Khoshgoftarr, Edward B. Allen, Robert Halstead, and Gary P. Trio, "Detection of Fault-Prone Software Modules During a Spiral Life Cycle", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 69-76.

[PEA95] Troy Pearse and Paul Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", Proceedings of the International Conference on Software Maintenance, Opio (Nice), France, October 17-20, 1995, pp. 295-303.

[PIG94] Thomas M. Pigoski and Lauren E. Nelson, "Software Maintenance Metrics: A Case Study", Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, September 19-23, 1994, pp. 392-401.

[SCH97] Norman F. Schneidewind, "Reliability Modeling for Safety Critical Software", IEEE Transactions on Reliability, Vol. 46, No.1, March 1997, pp.88-98.

[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.

[SNE96] Harry Sneed, "Modelling the Maintenance Process at Zurich Life Insurance", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 217-226.

[STA96] George E. Stark, "Measurements for Managing Software Maintenance", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 152-161.

[WEY95] Elaine J. Weyuker, "Using the Consequences of Failures for Testing and Reliability Assessment", Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, Washington, D.C., October 10-13, 1995, pp. 81-91.

Measuring and Evaluating the Stability of Maintenance Processes

Dr. Norman F. Schneidewind
Code SM/Ss
Naval Postgraduate School
Monterey, CA 93943, U.S.A.

Voice: (408) 656-2719
Fax : (408) 656-3407
Email: schneidewind@nps.navy.mil

OUTLINE

MAINTENANCE STABILITY CONCEPT

MAINTENANCE STABILITY ANALYSIS

DATA AND EXAMPLE APPLICATION

RELATIONSHIPS AMONG MAINTENANCE, RELIABILITY, TEST EFFORT, AND RISK

CONCLUSIONS

1. Definition of Maintenance Process *Stability*: If it is desirable that software attribute A increase (decrease) with time, then maintenance process M is *stable* with respect to A if A monotonically increases (decreases) with time t where t is time of release or test time, depending on the nature of A.

2. Definition of Maintenance Process *Instability*: If it is desirable that software attribute A increase (decrease) with time, then maintenance process M is *unstable* with respect to A if A monotonically decreases (increases) with time t where t is time of release or test time, depending on the nature of A.

When neither 1 nor 2 holds, we say that it is inconclusive as to whether M is stable or unstable. As a practical matter, if we are unable to conclude that a process is stable, we may at least be able to conclude that it is *not* unstable.

LONG TERM RELATIONSHIPS AND METRICS

The following relationships are analyzed and metrics are computed over a sequence of releases:

1. Mean Time to Failure (MTTF).

2. Total Failures normalized by KLOC Change to the Code.

3. Total Maintenance Test Time normalized by KLOC Change to the Code.

4. Remaining Failures normalized by KLOC Change to the Code.

5. Time to Next Failure.

6. Remaining Failures Risk Metric.

7. Time to Next Failure Risk Metric.

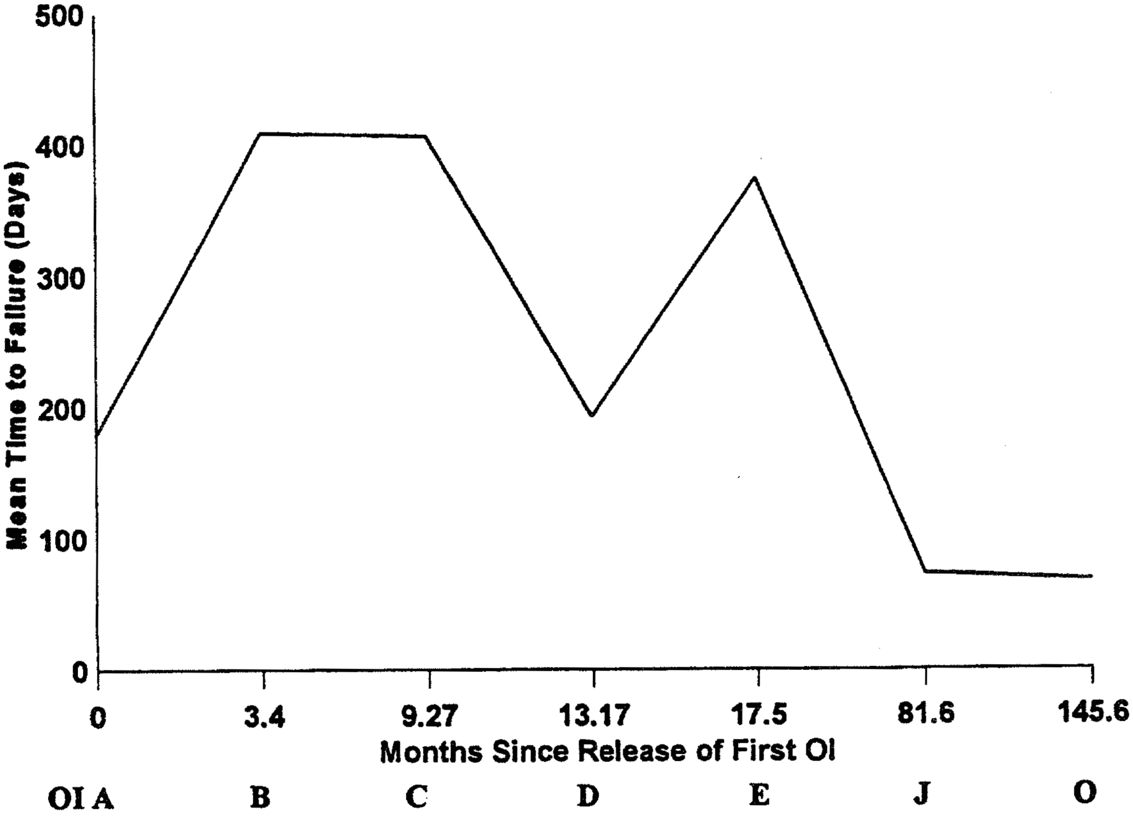
Table 1-Part 1: Characteristics of Maintained Software Across *Shuttle* Releases

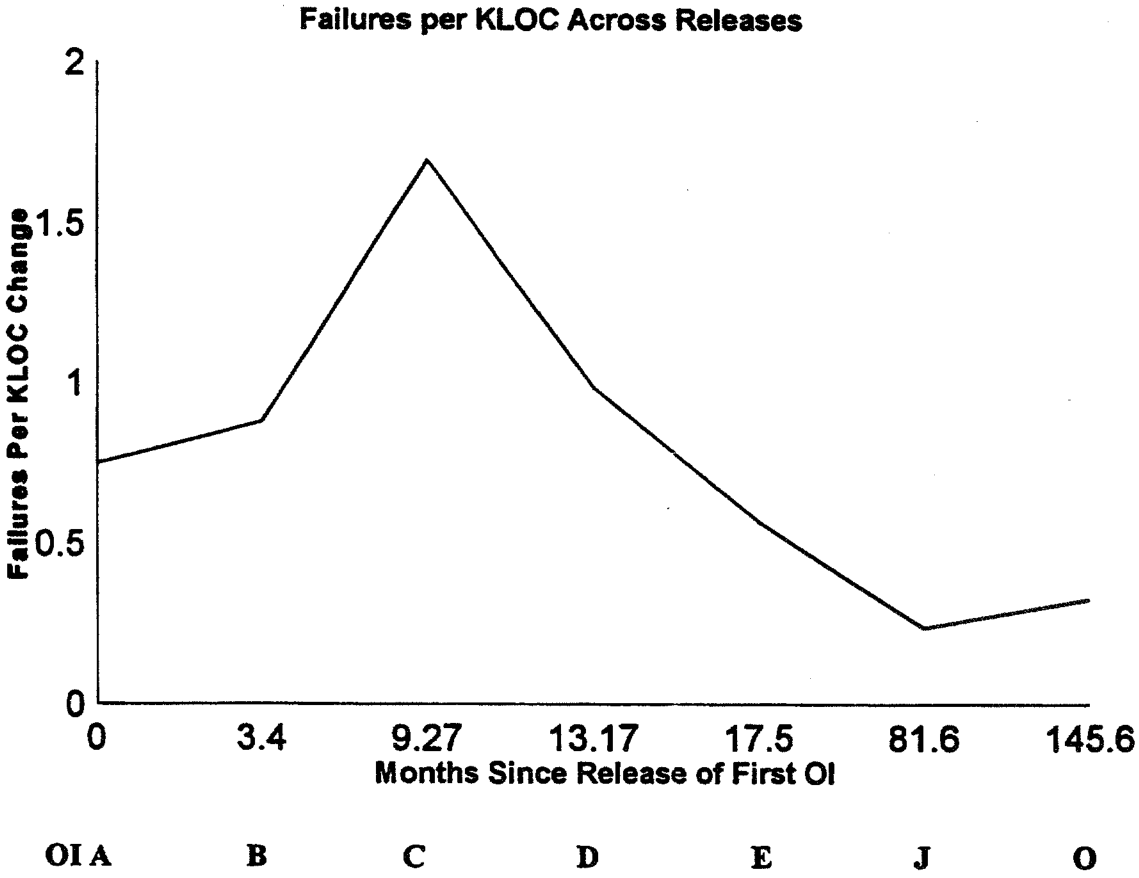
Operational Increment	Release Date	Launch Date	Mission Duration (Days)	Reliability Prediction Date	Total Post Delivery Failures	Failure Severity
A	9/1/83	No Flights		12/9/85	6	One 2 Five 3
B	12/12/83	8/30/84	6	8/14/84	10	Two 2 Eight 3
C	6/8/84	4/12/85	7	1/17/85	10	Two 2 Seven 3 One 4
D	10/5/84	11/26/85	7	10/22/85	12	Five 2 Seven 3
E	2/15/85	1/12/86	6	5/11/89	5	One 2N Four 3
F	12/17/85				2	Two 3
G	6/5/87				3	One 1 Two 3
H	10/13/88				3	Two 1N One 3
I	6/29/89				3	Three 3
J	6/18/90	8/2/91	9	7/19/91	7	Seven 3
K	5/2/91				1	One 1
L	6/15/92				3	One 1N One 2 One 3
M	7/15/93				1	One 3
N	7/13/94				1	One 3
O	10/18/95	11/19/96	18	9/26/96	5	One 2 Four 3
P	7/16/96				3	One 2 Two 3
Q	3/5/97				1	One 3

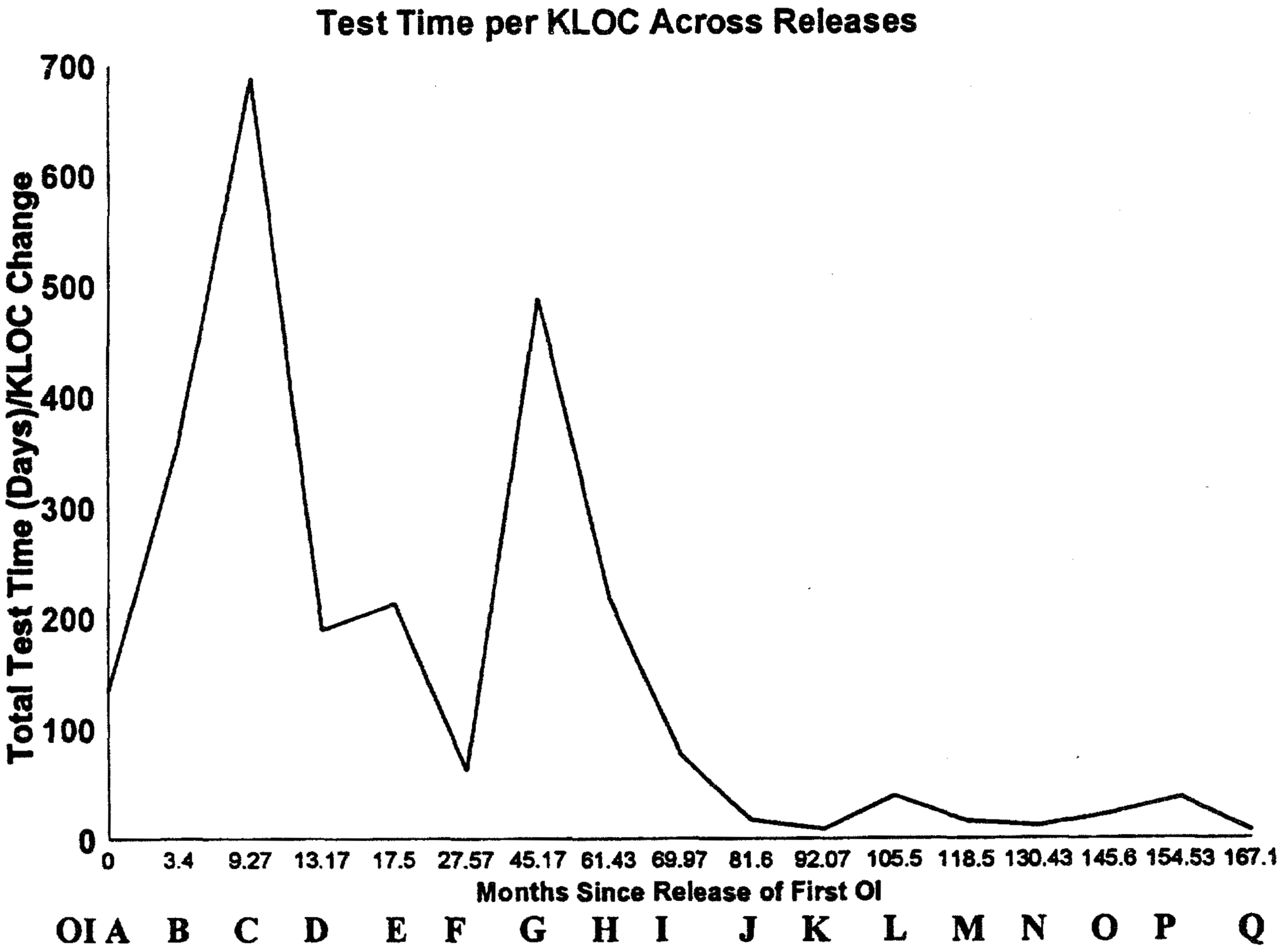
Table 1-Part 2: Characteristics of Maintained Software Across *Shuttle* Releases

Operational Increment	KLOC Change	Total Test Time (Days)	MTTF (Days)	Total Failures/ KLOC	Total Test Time/ KLOC
A	8.0	1078	179.7	0.750	134.8
B	11.4	4096	409.6	0.877	359.3
C	5.9	4060	406.0	1.695	688.1
D	12.2	2307	192.3	0.984	189.1
E	8.8	1873	374.6	0.568	212.8
F	6.6	412	206.0	0.300	62.4
G	6.3	3077	1025.7	0.476	488.4
H	7.0	540	180.0	0.429	77.1
I	12.1	2632	877.3	0.248	217.5
J	29.4	515	73.6	0.238	17.5
K	21.3	182			8.5
L	34.4	1337	445.7	0.087	38.9
M	24.0	386			16.1
N	10.4	121			11.6
O	15.3	344	68.8	0.327	22.5
P	7.3	272	90.0	0.411	37.3
Q	11.0	75			6.8

Mean Time To Failure Across Releases




Failures per KLOC Across Releases
Failures Per KLOC Change
2
1.5
1
0.5
0
0
3.4
9.27
13.17
17.5
81.6
145.6
Months Since Release of First OI
OI A
B
C
D
E
J
O


Test Time per KLOC Across Releases
Total Test Time (Days)/KLOC Change
700
600
500
400
300
200
100
0
0
3.4
9.27
13.17
17.5
27.57
45.17
61.43
69.97
81.6
92.07
105.5
118.5
130.43
145.6
154.53
167.1
Months Since Release of First OI
OI A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q

Reliability of Maintained Software -- Remaining Failures
Normalized by Change to Code

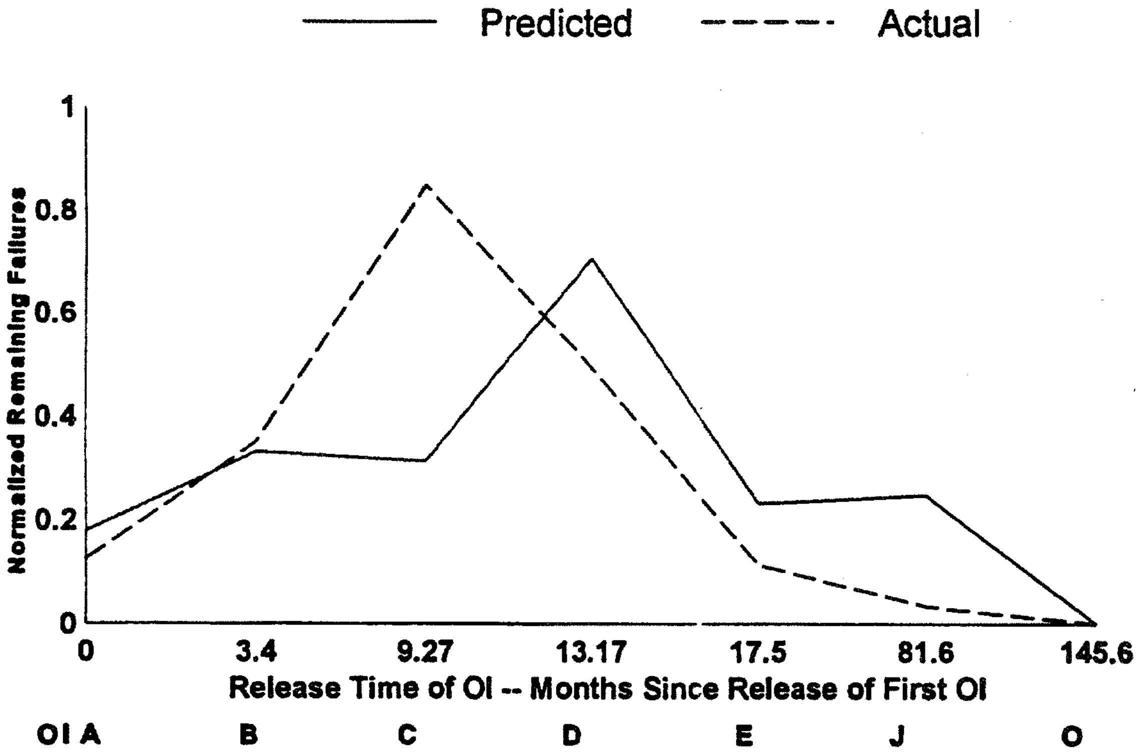
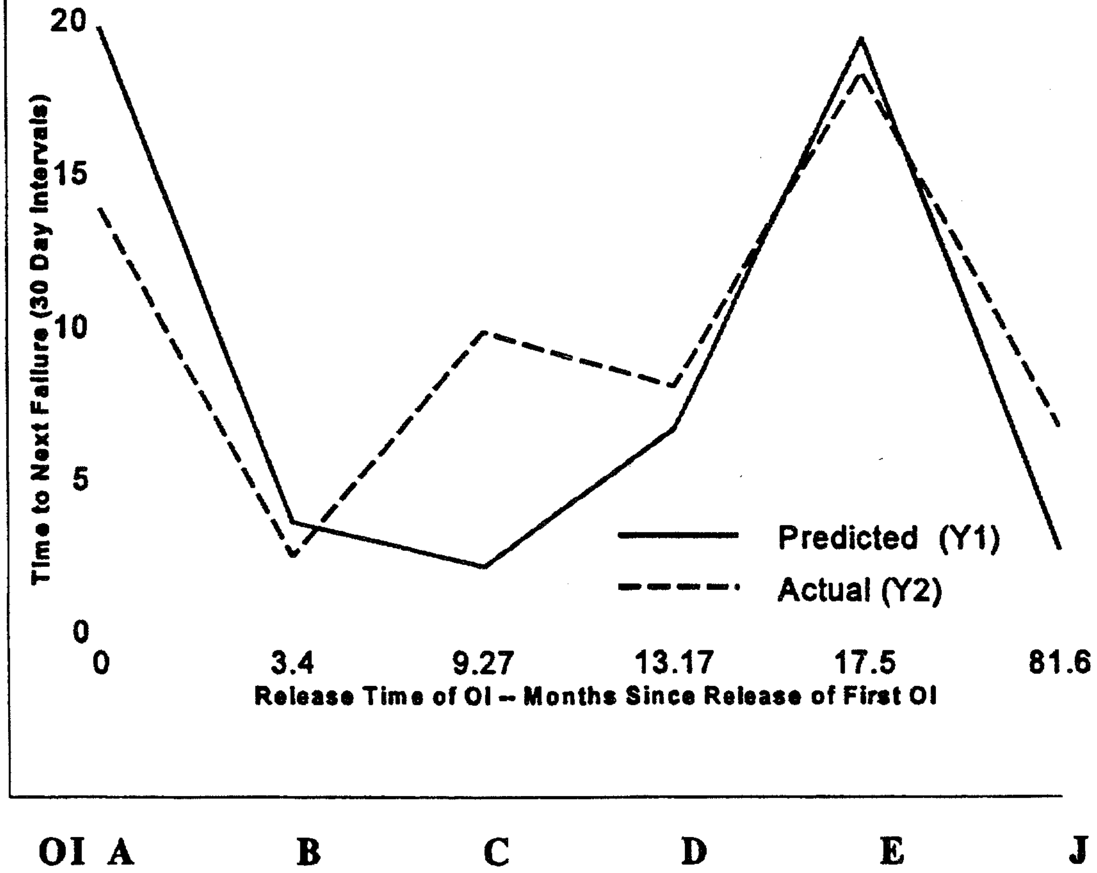
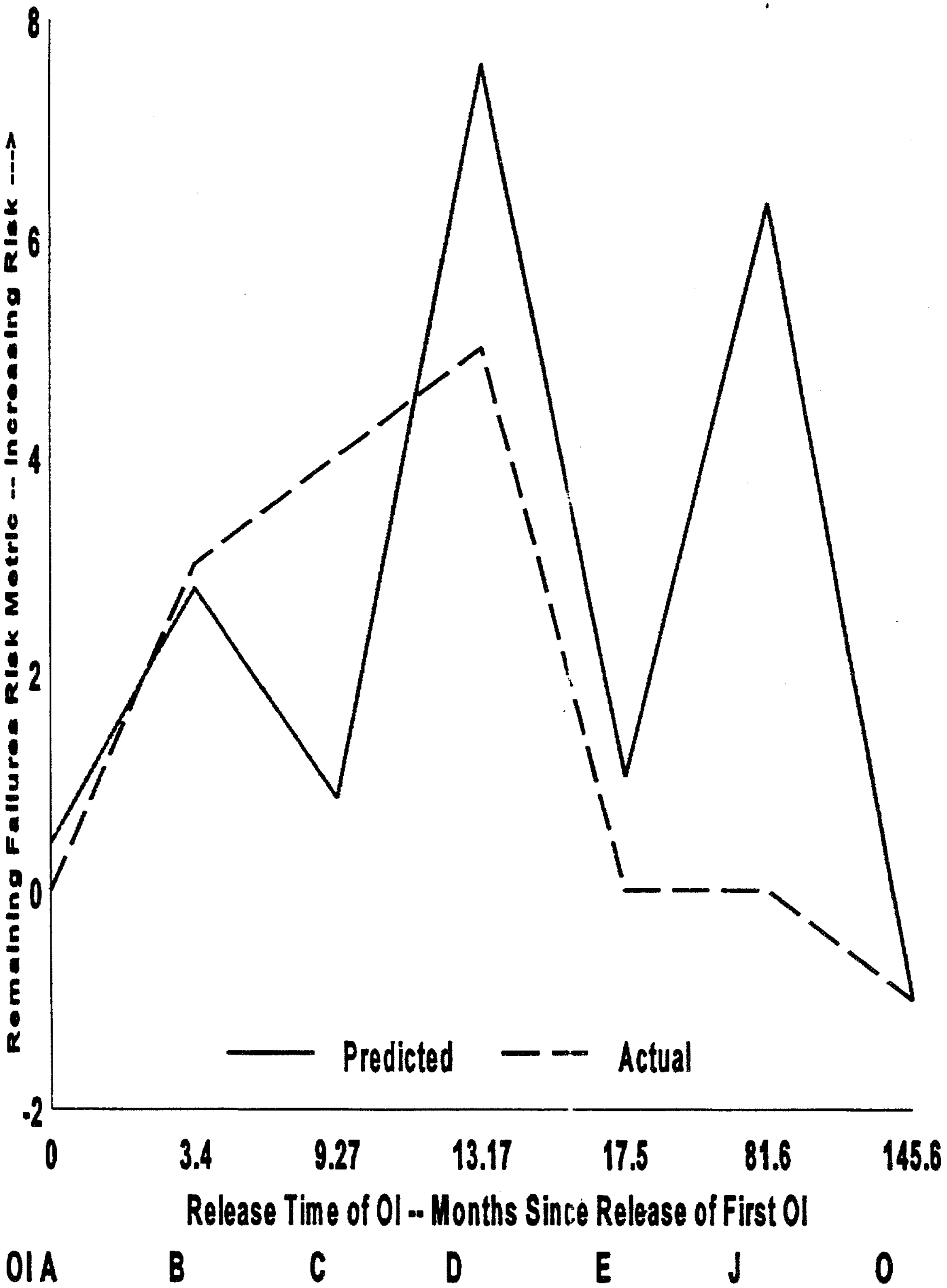


Figure 5. Reliability of Maintained Software -- Time to Next Failure



Remaining Failures Risk Metric for 1 Remaining Failure

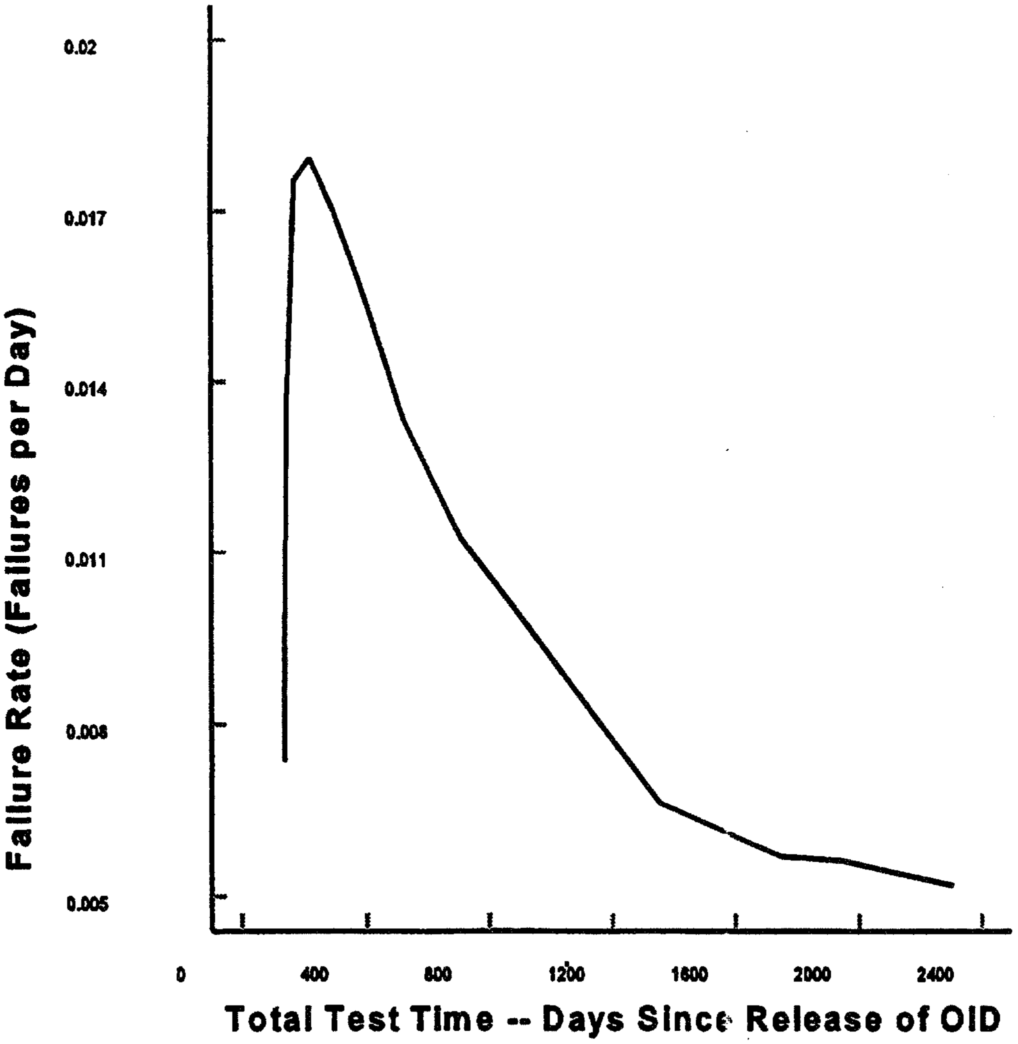


SHORT TERM RELATIONSHIPS AND METRICS

The following relationships are analyzed and metrics are computed within a given release:

1. Total Maintenance Test Time versus Number of Remaining Failures.

2. Failure Rate versus Total Test Time.



Reliability of Maintained Software: Failure Rate of OID

CONCLUSIONS

We conclude, based on both predictive and retrospective use of maintenance, reliability, test, and risk metrics, that it is feasible to measure the stability of a maintenance process and to integrate these factors into a unified approach for assessing the impact of the maintenance process on the reliability and risk of deploying the software.

In this research we were limited to using a qualitative approach to assessing the functionality and complexity of the changes to the Shuttle code.

A future research goal is to use various software characteristic metrics in conjunction with a larger sample of failure data that can be obtained by combining pre release and post release failure data.

Verification and Validation in a Rapid Software Development Process

John R. Callahan and Steve M. Easterbrook
NASA Software IV&V Facility
100 University Drive
Fairmont, WV 26554
304-367-8235
{callahan, steve}@ivv.nasa.gov

Abstract

The high cost of software production is driving development organizations to adopt more automated design and analysis methods such as rapid prototyping, computer-aided software engineering (CASE) tools, and high-level code generators. Even developers of safety-critical software systems have adopted many of these new methods while striving to achieve high levels of quality and reliability. While these new methods may enhance productivity and quality in many cases, we examine some of the risks involved in the use of new methods in safety-critical contexts. We examine a case study involving the use of a CASE tool that automatically generates code from high-level system designs. We show that while high-level testing on the system structure is highly desirable, significant risks exist in the automatically generated code and in re-validating releases of the generated code after subsequent design changes. We identify these risks and suggest process improvements that retain the advantages of rapid, automated development methods within the quality and reliability contexts of safety-critical projects.

1. Introduction

Rapid software development, or rapid application development (RAD), is a broad term characterized by the use of domain-specific computer-aided software engineering (CASE) tools in an iterative process development lifecycle to achieve functional software within short production schedules [1]. First, a basic design is sketched out as a collection of interconnected components using various structured methods . This step defines a basic architecture for a system of interconnected components. Next, the behaviors of some these components and their interactions are defined and implemented. These selected behaviors, often called *features*, are selected on the basis of their priority and utility relative to system requirements. When the selected features have been implemented, the system can be executed (either through simulation mechanisms or code generators) and tested within the scope of the implemented behaviors. Finally, the process repeats itself by enhancing the architecture and implementing the next set of selected features.

Many RAD organizations rely on separate testing group to exercise each partially functional release of the system. In general, these organizations are comprised of two separate but equal subgroups: a "design" group that is responsible for construction of each release and a "test" group that finds problems in each release and works with the design group to fix them. Many errors may remain in each release and it is the task of the test group to find and fix these problems quickly. The test group is responsible for working with the development organization to build revisions to the release.

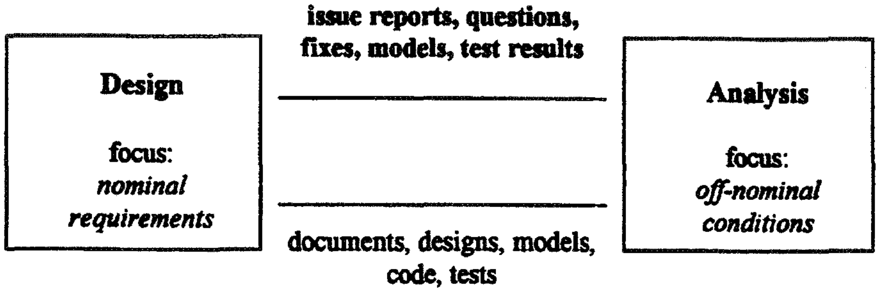

issue reports, questions, fixes, models, test results
Design
focus: nominal requirements
Analysis
focus: off-nominal conditions
documents, designs, models, code, tests

Figure 1: The dialectic between design and analysis groups

The design and test group jointly consider a release to be *stable* if most of the serious problems have been fixed. The process should repeats itself only when a stable release has been achieved. The inability to achieve such stability is an indication of a serious problem (e.g., an incorrect architecture or a poorly designed and implemented feature). Furthermore, the smooth execution of this bipartite process (Figure 1) is essential to the rapid development of software in such organizations. The focus of the design group is typically on nominal behaviors of the system. A nominal behavior is any feature that exhibits an error-free execution of the system. In contrast, the focus of the test group is on off-nominal behaviors of the system. Their analysis should include examination of feature interactions, faults, and unexpected inputs. As organizations attempt to achieve "rapid" development (i.e., deadlines are compressed), this bipartite model becomes important because the focus of the design group tends to become increasingly myopic toward nominal behaviors. The complementary role of the test group usually offsets this tendency and provides a corrective force to the process.

This "build-and-smoke" [2] or "synch-and-stabilize" [3] approach to development is practiced currently in large software development companies because it can deliver functional but incomplete software quickly. Many problems, however, can arise during the process. If the test group does not possess the necessary analysis skills to perform their task, then their contribution is diminished or even dismissed. On the other hand, if the role of the test group is misunderstood by the design group or management does not accept the findings of the test group, then the dialectic between the two groups is pathological. It is management's responsibility to keep the channels of communication open and limited to constructive criticism.

This paper examines a specific case study in which a very large software development organization must interact with an independent verification and validation (IV&V) contractor to achieve incremental, stable releases of software subsystems for the International Space Station (ISS) project. The development contractor plays the role of the design group. They are responsible for the production of stable software releases for ISS subsystems. The IV&V contractor plays the role of test group but applies many types of analysis to the system design and implementation in order to "test" each release.

2. Verification and Validation

IV&V is a *systems engineering* discipline that applies many technical analysis and testing methods to various development artifacts and processes during all phases of the software development lifecycle [4]. Verification is any analysis activity that tries to demonstrate that the product of a phase during development is consistent and complete with respect to the specification before that development phase. Validation is any analysis activity that tries to demonstrate that the product of any development phase is consistent with domain and application requirements. Both of these activities are important during software development since each phase introduces transformations that may be incorrect with respect to the specifications or the intended utility of the overall system.

A V&V organization can be independent with respect to its technical, financial, and managerial relationships with the design group. A technically independent V&V group uses different tools and techniques than the design group to analyze project artifacts throughout development. A financially independent V&V group may be funded an external quality assurance group or oversight body. A managerially independent V&V group usually reports to the customer or the person above the supervisor of the design group. In general, since independence mostly involves the organizational aspects of analysis, we will frequently use the term "V&V" instead of "IV&V" to describe the analysis activity itself.

V&V analysts also play an important role during maintenance because a significant portion of maintenance tasks involve functional enhancements to the behavior, design, and implementation of a system [5]. V&V's primary task is to manage project risk by identifying and monitoring errors throughout the development and maintenance process. Since it is impossible to identify and resolve all errors early in a project's lifecycle, it is the V&V contractor's task to identify errors as early as possible and track the progress towards their resolution. For example, a minor design error may be ignored early in the process if the developer believes that a yet-to-be-designed feature will solve the problem.

During development and maintenance, IV&V maintains a list of reports on problems found during the development process [6, 7]. It tries to verify solutions to these problems and produces reports on new problems when necessary. Such reports come in a wide variety of formats an include items such as change requests (CRs), discrepancy reports (DRs), problem reports (PRs), issue reports (IRs), and issue tracking requests (ITRs). Most of these reports are authored by an IV&V contractor during development but can originate in the design group as well. Each report has a disposition that changes as a problem is addressed throughout development. V&V will track problem reports and ensure that each report is eventually addressed at an appropriate point during development. If the problem is not adequately addressed, then V&V can report this up the management hierarchy. In most cases, however, this route is avoided. Most problem reports remain as part of the normal dialogue in confidence between the design and V&V groups.

3. Case Study

Exploration of space requires the use of sophisticated software with high levels of quality and reliability. On the International Space Station (ISS) project, one contractor decided to use the MatrixX[1] tool to reduce development costs and improve design quality. The tool was used to develop and perform white-box testing (unit and integrated component level verification) of human-rated critical flight software. The tool is extremely useful in designing and generating code for complex systems. Our task was to identify process issues related to the tool's use in the ISS effort and suggest paths for achieving the highest possible quality and reliability via testing during the development process.

Figure 2 is a high-level conceptual model of the production process for each software release. The ISS software design is comprised of several computer software configuration items (CSCIs) onto

[1] MatrixX is a trademark of Integrated Systems Inc.

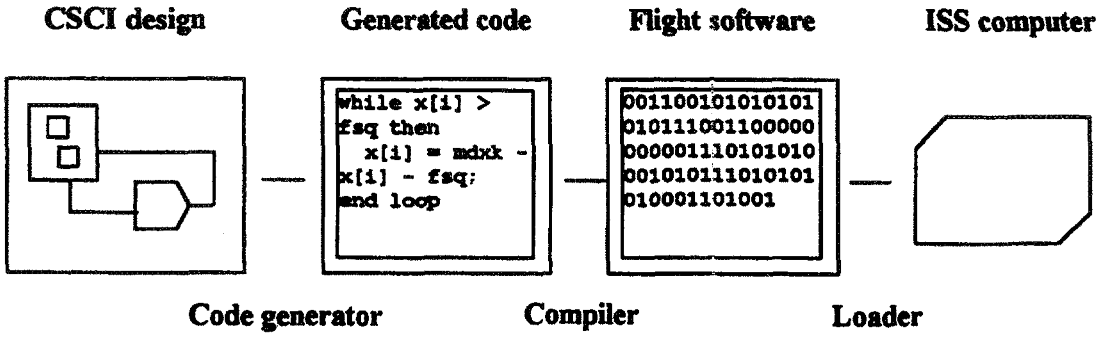


Figure 2: Release production process

which are loaded one or more computer software components (CSCs). Each computer software component is designed as a hierarchically nested set of computer software units (CSUs) that can be tested in isolation through simulation in the design tool. Source code (in Ada) can be generated automatically for an entire CSCI using a code generation tool. The generated code is then compiled to the platform using a conventional compiler to produce the executable flight software and loaded onto a flight computer.

The contractor employs the design tool in a rapid software development process that calls for iterative design and testing of the high-level design and releases of the automatically generated code for each CSCI. Testing was planned for only the high-level CSCI design through simulation (a capability of the design tool) and the code generated for the entire system, but not of the automatically generated code at unit level. This was felt to be too expensive and redundant. Critical CSUs would also be tested at the unit level through simulation in the design tool. While the ability to simulate the design at the CSCI and CSU levels is extremely useful and can lead to the discovery of early design flaws, some errors can only be detected through analysis of the generated code:

- The code generator is not a certified, verified tool. As with any new tool, documented flaws have been shown to exist in the code generator. These flaws lead to errors in the translation process and could introduce problems in the source code that do not exist in the high level design. While visual code audits of the generated code may be useful, they are expensive and poor substitutes for unit testing.
- Important objectives of unit testing are to ensure that adequate test coverage is achieved and that extreme and singular values for variables are tested. Unit testing of the high level design cannot achieve this in generated code that may introduce auxiliary and temporary tasks, procedures,, and variables.
- For any number of reasons (data storage, real-time processing, etc.), the design environment may not behave as the actual environment will under identical test inputs. It cannot be assumed that they will behave similarly in all situations.

The developer argued that while these concerns were valid, they did not represent significant risk to the project when balanced against the productivity and design quality gains. The V&V contractor, however, argued that these were significant risks that must be addressed in the short and long term. It was our task, as a research team, to help identify the issues while seeking to preserve use of the code generator without undue impact on project costs, schedule, and functionality.

3.1 Problems

The use of code generators to produce flight software creates several testing and maintenance problems. The problem originally focused on unit testing aspects of the generated code, but quickly expanded to other areas of concern:

- The sequencing characteristics of the code generator creates problems related to long-term maintenance of the software. When changes to the software design are necessary, they will be done in the high level specification. Since the code generator uses data flow analysis to produce a single, interleaved module for an entire system, small design changes are highly likely to produce significant changes in the entire body of generated code. As a result, a significant of amount of regression testing of the generated system will be required for even small changes. The cost of this additional testing threatens to erode cost savings realized by use of the tool in the first place.
- The code generator makes any errors found during testing and mission operations difficult to isolate and debug. Errors caused by problems other than design flaws cannot be debugged in the high level design. Problems other than design flaws may require changes to the generated code itself (e.g., such cases did occur in practice on this project). Such changes create a divergence from the design and enormous configuration control problems in the short and long term.
- The automatically generated code does not comply with Ada coding standards. While the tool vendor never intended for generated code to be read or altered, *such changes are occasionally necessary to access functions not available in the high-level design environment.* Furthermore, readability of the generated code is desirable because the system much often be debugged at this level. Thus, some form of structuring and coding standards are necessary to ensure readability, maintainability, reliability, reusability, and portability. It is highly unlikely that over the entire lifecycle of the system, the code will remain unexamined. Based on the size of the overall project and experience with other large flight software projects, it will become necessary at some point to examine the generated code.
- The code generator currently produces inefficient code in terms of size and memory usage. While the manufacturer is improving the technology of the tool (a problem itself - see below), it was estimated that the ISS GN&C will generate 12,382 SLOCs requiring 386K for storage. The current design produces over 120,000 SLOCs and would have required secondary storage in addition to the original 1MB EEPROM for each MDM. Before a recent redesign initiated by the identification of this problem, this code bloat may have dictated a significant hardware change that would have significantly increased system fault risks (i.e., seek delays and potential failures of the secondary storage unit).
- There existed no satisfactory configuration control plan for long-term evolution of the design specification, generated code, testing, and new releases of the code generator tools. Plans and processes need to be developed with regard to upgrading to new tool releases, how this affects code generation, unit and system level testing.

These problems are likely to occur with the use of any high level design and code generation tools including modern programming language compilers, linkers, and loaders. It was our task to help identify these issues and develop approaches to mitigate risk while leveraging the advantages of the tools.

3.2 Solutions

Based on these findings, the development and V&V contractors worked together to develop a comprehensive plan that leverages the productivity gains of the tool while integrating quality and reliability goals. Some of the adopted plans include:

- Heavy use of generated code modularization. The code generator contains features for isolating the generation of code for specific design units into modules. This feature introduces code bloat, but significantly reduces coupling in the generated code.
- Unit testing of generated code. It was decided that the modularization features made it possible to effectively unit test the generated code. This includes assessment of code coverage adequacy based on path/logic coverage, data minimum/maximum values, and erroneous data inputs.
- Auto-generation of test cases. One additional benefit of code modularization was that tests on the design specification could now be used to generate system-level test cases.
- Adoption of a configuration control, test, upgrade and integration plan. This plan includes a modest upgrade path combined with regression tests for modules that are expected to be affected by changes to the code generator and design changes. The reduction in coupling means that newly integrated design modules no longer have ripple effect on the rest of the generated code and can be tested in isolation from the rest of the system specification.

Our recommendations focused on the use of design modularization features to achieve and maintain fidelity between the design and generated code. Although not easy to use and not enforced at the design level, the modularization conventions have been extremely useful in facilitating the iterative design and analysis process. Indeed, the tool manufacturer is planning to incorporate enforcement of modularization in subsequent releases of the tool.

4. Summary

Automated tools will continue to be used with increasing frequency in software development projects for many good reasons including cost, quality, and productivity. Indeed, our analysis supports the continued use of CASE tools, but we must be continuously aware of the risks associated with new technologies that are co-evolving with our projects. We found that it is possible to integrate safety-critical goals of quality and reliability during the development process if existing organizations include complementary advocates for nominal behaviors (the designers who want to see the software achieve specified functionality) and advocates for off-nominal behaviors (the V&V team who want to ensure that rare cases have been accounted for as best as possible). If both teams work together within an iterative process that facilitates both design and analysis, then concomitant goals can be achieved.

Many of our suggested improvements were based on the need to reduce process risks as well as reducing the risk of errors in the product. The benefits of V&V analysis can only be leveraged if the turn-around time for analysis can be streamlined. The modularization and configuration control plans greatly enable V&V to provide timely and useful analysis to the design group. Our recommendations helped reduce the tremendous amount of rework that would have been necessary to maintain a productive dialogue between the two groups.

Finally, it is V&V's continual task to monitor the co-evolution of CASE tools used on the project by analyzing the differences in generated code between tool versions. Analysis of these differences will provide useful information for tailoring the verification process to accommodate known differences between the design and deployment environments. Depending on changes to the tool, extensive regression tests may be necessary for some project releases because of the impact on the generated code regardless of modularization boundaries.

5. Acknowledgements

We would like to thank Tom Marhsall, Jim Dabney, and Dan McCaugherty of Intermetrics, Inc. for their help in collecting and summarizing this data. We are most grateful for their time, hard work, and dedication to quality performance of V&V tasks. This report has been prepared under NASA Cooperative Agreement NAG2-797 through funding from NASA's Office of Safety and Mission Assurance and the NASA/Ames Research Center.

6. References

1. McConnell, S., *Rapid Development: Taming Wild Software Schedules*. 1996, Redmond, WA: Microsoft Press.
2. McConnell, *Daily Smoke and Build Test*. IEEE Software, 1996. 13(4): p. 144.
3. Cusumano, M. and R. Selby, *Microsoft Secrets*. 1995: The Free Press. 512.
4. Lewis, R., *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*. 1992, New York: John Wiley & Sons. 356.
5. Garlan, D., G.E. Kaiser, and D. Notkin, *Using Tool Abstraction to Compose Systems*. j-COMPUTER, 1992. 25(6): p. 30-38.
6. Callahan, J. and G. Sabolish. *A Process Improvement Model for Software Verification and Validation*. in *The 19th Software Engineering Workshop*. 1994. NASA Goddard Space Flight Center, Greenbelt, Md.
7. Callahan, J., T. Zhou, and R. Wood. *Software Risk Management through Independent Verification and Validation*. in *4th International Conference on Software Quality (ICSQ 94)*. 1994. McLean, Va.

Verification and Validation in a Rapid Software Development Process

John R. Callahan

http://research.ivv.nasa.gov/

NASA Independent Software V&V Facility Software Research Lab

West Virginia University

Department of Computer Science & Electrical Engineering

Overview

- Rapid Software Development
- Independent Verification and Validation
- A Bipartite Model of Software Development
- Case Study
  - Problems
  - Solutions
- Conclusions

Rapid Software Development

- Also called Rapid Application Development (RAD)
- A process characterized by:
  - the use of automated design tools (e.g. CASE)
  - a short production schedule of successive releases with limited features and enhancements
  - an iterative, evolutionary lifecycle in which the requirements, design and implementation may change

[McConnell96]

Build 'n Smoke OR Sync 'n Stabilize

- Many RSD/RAD organizations are composed of two distinct and complementary subgroups:
  - DESIGN ("the builders")
    - responsible for system requirements, design, implementation and testing of *nominal* functionality
  - ANALYSIS ("the breakers")
    - responsible for analysis of requirements, design, implementation and testing of nominal and *off-nominal* behaviors
    - plays a constructively critical role in development (not adversarial)
    -

[Microsoft Secrets, CusSel95]

Independent V&V (called IV&V)

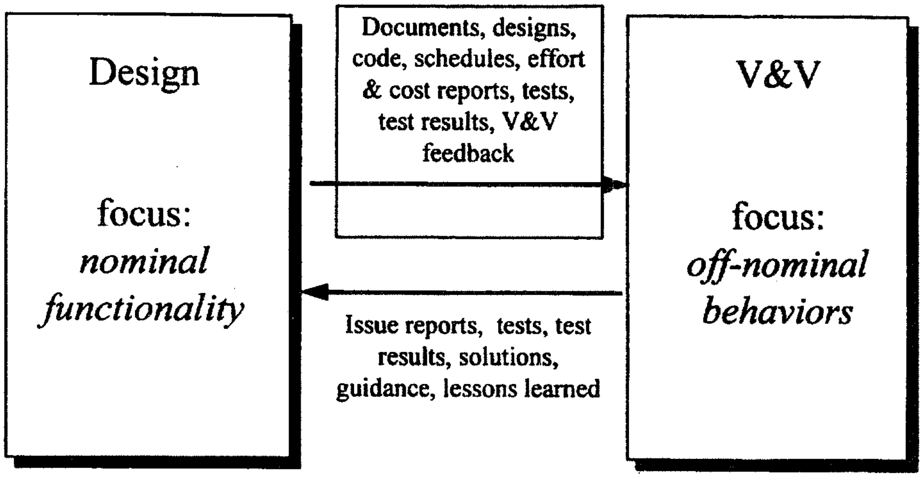
- A systems engineering discipline that applies various forms of analysis at all phases of development
- Verification
  - Ensure that the output of each phase is consistent with the input specifications
  - *ARE WE BUILDING THE PRODUCT RIGHT?*
- Validation
  - Ensure that the output of a phase is consistent with the requirements and domain constraints
  - *ARE WE BUILDING THE RIGHT PRODUCT?*
- Independence: managerial, financial, technical

[NRC Report, Leveson 93]

V&V in Rapid Development

- Problems in large, complex multi-vendor projects
  - Requirements not fully understood
  - Design in flux due to technological advances
  - New automated tools and processes in use
  - High turnover and staffing problems
  - Limited customer visibility
- The focus of the DESIGN group becomes myopic w.r.t. nominal behaviors of the system under construction as schedules compress
- An independent advocate/analyst of off-nominal behaviors (e.g., faults, unexpected conditions) is needed to complement this bias

A Bipartite Model of Software Development

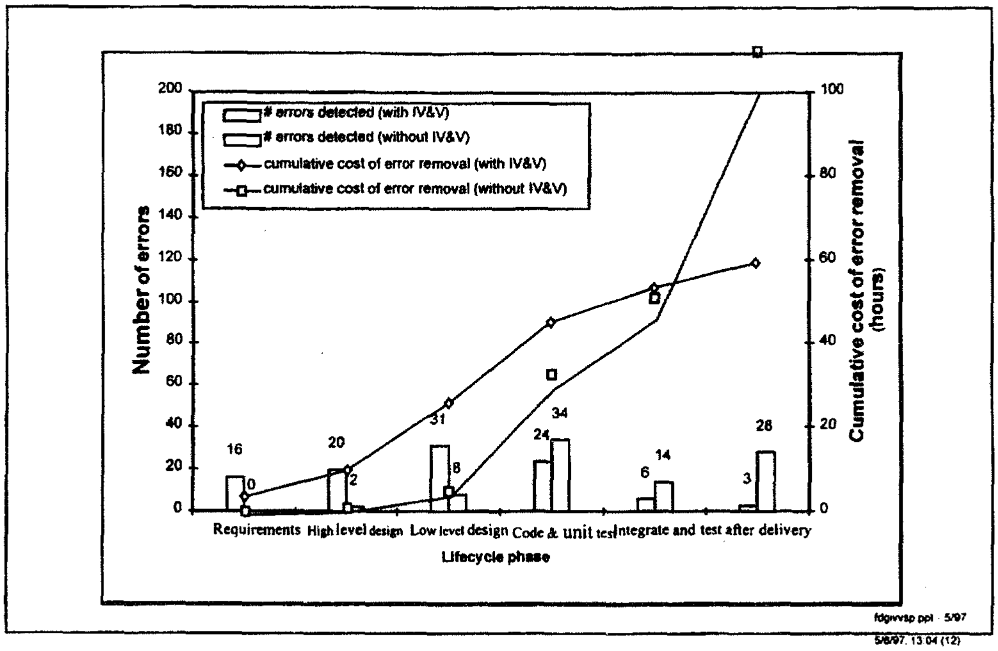

Design
focus:
nominal functionality
Documents, designs, code, schedules, effort & cost reports, tests, test results, V&V feedback
Issue reports, tests, test results, solutions, guidance, lessons learned
V&V
focus:
off-nominal behaviors

Issue Reporting

- An issue report contains details of an inconsistency identified during the development process
- An issue report contains information such as:
  - *a description of the problem*
  - *phase in which the problem was found*
  - *the criticality of the problem*
  - *proposed solutions and options*
- Issues are best reported directly to the design group (NOT through management unless necessary)
- Issue databases must be managed, tracked, and statistically analyzed for trends

[Easterbrook & Callahan 96]

IV&V Effectiveness



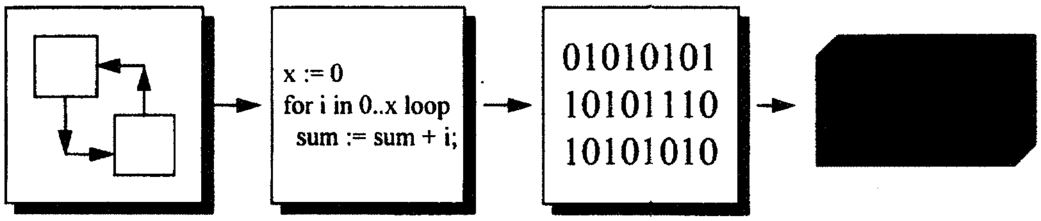
fdgivvsp.ppt 5/97
5/8/97, 13:04 (12)

Case Study

- An issue report involving the use of the MatrixX* CASE tool on the International Space Station (ISS) flight software development project
- Issue: unit testing on generated flight software (FSW) code (in Ada)
- Issue was contested by design group who claimed that unit testing within the design tool (through simulation) was adequate
- The high criticality of the issue and conflict of opinion prompted management involvement
- Research team asked to examine the issue in detail

Overview of Autocode Process

- MatrixX provides an environment for designing systems (CSCIs) composed of connected blocks of subsystems
- An autocoder is used to generate code for a complete CSCI
- A conventional compiler is used to produce flight code from the generated Ada source



Problems

- Testability and maintainability of FSW
- Increased difficulty debugging problems
- Code generator problems and inefficiencies
- Lack of configuration control plan for future tool improvements

Sub-issue #1

- Unit testing of FSW components is necessary to reduce the amount of system-level testing needed to maintain release schedule
- The autocoder uses control and data flow analysis of the block design to generate interleaved code
- Subsystem boundaries disappear in the generated code due to extensive interleaving
- Extensive regression testing of generated FSW will be necessary because small changes to the design are highly likely to propagate throughout the generated code

Sub-issue #2

- Errors found during testing and mission operations are difficult to isolate and debug
- The interleaved nature of the generated code makes it difficult to trace problem in deployed code to the high-level design
- Such problems have prompted the need for direct changes to the generated code
- This creates enormous configuration control problems for development

Sub-issue #3

- The generated code does not comply with Ada coding standards
- While the generated code was never meant to be read or understood
- BUT, it is highly likely that at some point during the project's duration, someone will need to examine the generate source code
- Documented problems exist in the code generator due to the immaturity of the technology

Solutions

- Use modularization mechanisms in MatrixX to control generation of code and preserve unit boundaries
- This leads to some inefficiencies in code size and CPU usage
- Limits on feature development in releases help offset code bloat and processor overhead caused by modularization
- Working with tool vendor to improve code generation and test support mechanisms
- These solutions reduce turn-around time for V&V analysis by reducing system-level tests on generated FSW code and enabling unit testing

Conclusions

- Highly advocate the continued use of MatrixX
- High-level design and simulation are beneficial to productivity and quality
- V&V is an essential part of rapid development itself
- V&V is not just an add-on "insurance"
- Rapid development processes need some form of iterative analysis to guide system design evolution
- More empirical studies of the dialectic between design and analysis are needed
- Collection of issues made easier through automated tools and email

For more info...

http://research.ivv.nasa.gov/
http://www.ivv.nasa.gov/

Using Semantic Distance to Make Adaptation Decisions: Extended Abstract

Lamia Labed Jilani
Regional Institute for Research in
Computing and Telecommunbications
Cite Montplaisir,
Belvedere 1002 Tunisia
Email: lamia.labed@irsit.rnrt.tn
Fax: (216) 1 787 827

Ali Mili
West Virginia University and
Institute for Software Research
1000 Technology Drive
Fairmont, WV, 26554, USA
Email: amili@cs.wvu.edu
URL: http://www.isr.wvu.edu/

September 7, 1997

1 Adaptation Decisions

In this extended abstract, we report on an experiment that we are conducting to derive a model that supports decision making in program adaptation. We refer to two specific decisions, which are closely related yet distinct:

- Given a software specification K and a software component C that *almost* satisfies K (in a sense to be defined), is it best (most economical) to adapt C to satisfy K or to build a solution to K from scratch?
- Given a software specification K and two software components C and C' that *almost* satisfy K, which of C and C' is a better adaptation candidate (costs less to adapt)?

In principle at least, the first question is a special case of the second, since it amounts to comparing the available component C against the trivial (empty) program. Hence we will focus our attention on the second question, and discuss later to what extent our findings apply to the first question.

2 Syntactic and Semantic Distances

Given two specifications (or programs) R and R', it is possible to distinguish between two types of measures of distance: *syntactic* (or: structural) distance, which reflects to what extent R and R' *look alike*; and semantic (or: functional) distance, which reflects to what extent R and R' *act alike*.

For the sake of adaptation effort, it is best to use syntactic distance than to use semantic distance: when two specifications (or programs) look alike, then it is easy to adapt one to obtain the other. The trouble with synrtatic distance, however, is that it is difficult to estimate ahead of time when all we are given are a specification R and an adaptation candidate C: In order to estimate the distance between K and C, we must be able to determine whether C looks like a possible solution to K —not an easy task unless we derive a solution to K, which defeats the purpose of the exercise.

Hence we fall back on using measures of semantic (functional) distance. This approach is based on the premise that if the function of a program and that of a specification are similar, then it is easy to adapt the program to satisfy the specification. While one would hope this holds in general, there are counter examples where it does not: an insertion sort and a quicksort perform the same function but do not look alike; on the other hand, a program that performs the sum of an array and a program that performs the product of an array look alike but do not act alike (hardly ever produce the same result for a given array). Such pathological examples notwithstanding, one would hope that in general, functional distance is a reasonable indication of structural distance (hence of adaptation effort).

3 Measures of Functional Distance

We have defined six measures of functional distance between relational specifications; these measures are based on mathematical formulas and can be evaluated and compared by means of theorem provers. The measures differ by the aspects of similarity or dissimilarity that they reflect between the given specifications, and are briefly presented below:

- *Functional Consensus.* This measure reflects the amount of information that its arguments have in common; it is useful for the sake of adaptation because the more information a program and a specification have in common, the more features of the specification are already covered by the program.

- *Refinement Difference.* This measure reflects the amount of functional requirements of the specification that are not covered by the program; it is useful because the smaller this amount, the less one needs to add to the program to satisfy the specification.

- *Functional Excess.* This measure reflects the amount of functional features of the program that are irrelevant to the requirements of the specification; it is useful because the less such features the program has, the less distractions the programmer will face when adapting the program to satisfy the specification.

- *Refinement Distance.* This measure adds up (in a lattice-theoretic sense) refinement difference and functional excess, and can be justified by means of the justifications given above.

- *Functional Tangent.* This measure is a vector whose first entry (called its numerator) is functional excess and whose second entry (called its denominator) is functional consensus; it is useful because it attempts to maximize functional features of the specification that are already covered by the program, while minimizing functional features of the program that irrelevant to the specification.

- *Refinement Ratio.* This measure is also a vector whose numerator is refinement distance and whose denominator is functional consensus; it is useful because it maximizes common information while minimizing discriminating information.

These measures of distance take their value, not in the set of non-negative real numbers (as traditional measures of distance), but rather in the set of relational specifications. The set of relational specifications is partially ordered (by the refinement ordering), hence giving us means to compare distances —if only partially.

4 Correlating Syntactic and Semantic Distances

In order to investigate possible correlations between syntatic and semantic distance, we have run the following experiment.

- We consider a software library that consists of twelve software components, $C_1..C_{12}$, and a set of fourteen queries, $K_1..K_{14}$.

- For each measure of functional distance (there are six of those), and for each query $K_i, 1 \leq i \leq 14$, we compute all twelve measures of distance $d(K_i, C_j), 1 \leq j \leq 12$, and we compare distances between the components and K_i. For each measure of distance and each value of K_i, we draw a graph that shows how the components compare with respect to their proximity to K_i.

- For each query K_i and each component $C_j, 1 \leq j \leq 12$, we determine analytically how much each of the building blocks of C_j needs to be modified to accomodate specification K_i. From this analysis, we derive, for each specification K_i, a graph that shows how components C_j compare with respect to the effort required to adapt them to satisfy K_i.

- By assessing, for each K_i, to what extent the graph derived for adaptation effort is similar to the graphs derived for each of the six measures of distance we can determine how well each measure of functional distance would help to predict adaptation effort.

- By averaging the above results over all the K_i, we get a ranking of measures of distance.

This experiment requires proving hundreds of theorems, stemming from evaluating the measures of distance and comparing them. We have depended on a theorem prover for these, but did have to do many by hand as well. We compared graphs that stem from the measures of functional distance to the graphs that stem from adaptation effort using two criteria: first to what extent the graphs derived from each measure of distance produce the same optimal elements as those shown by the graph of adaptation effort; and second to what extent the graphs derived from each measure of distance look like the graph derived from adaptation efforts. The results show a consistent pattern whereby some measures of distance perform consistently better than others; also some measures achieve a high degree of precision (0.63) and recall (0.79) in retrieving the components that prove to be optimal by the criterion of adaptation effort.

5 Extensions

We envisage to repeat the experiment with other software libraries, hence, obviously, other sets of queries as well, and see if the ranking of measures of distance is preserved. Also, we wish to consider the question of whether the selected measure of functional distance can be used to make a reuse vs develop-from-scratch decision, by merely comparing $d(K, C)$ and $d(K, \emptyset)$ for the selected measure of distance d —it sounds too good to be true!

Using Semantic Distance to Make Adaptation Decisions

L. Labed Jilani, IRSIT, Tunisia
A. Mili, The Institute for Software Research, USA

November 12, 1997

**22nd Annual Software Engineering Workshop
Greenbelt, MD December 3-4, 1997**

1 Adaptation Decisions

Two Adaptation Decisions:

- Adapt or develop from scratch? By inspection of K and C.
- If we choose to adapt, how do we select optimal adaptation candidate? By inspection of K, C and C'.

Existing Models

- COCOMO '81, EDSI metric and equations.
- COCOMO 2.0 reuse model, ESLOC metric and equations.

Economic Models

- Produce decisions, but also quantifications (budgeting, planning, etc).
- Rely on a great deal of expertise, experience, reasoning by analogy, design decisions.

Proposed Alternative: Measures of Distance.

- Produce decisions, but no quantifications.
- Based on plain inspection of K and C.

Principle: Measuring distance between specifications.

- Adaptation decision: comparing $\delta(K, C)$ to $\delta(K,)$.
- Choosing adaptation candidate: comparing $\delta(K, C)$ and $\delta(K, C')$.

Two types of distances:

- **Structural Distance**: How much query and candidate look alike. Syntax.
- **Functional Distance**: How much query and candidate act alike. Semantics.

Potentially Orthogonal.

- Sum and Product of an Array.
- QuickSort and InsertionSort.

Minimizing Adaptation Effort: Structural Distance. Dilemna:

- Structural distance cannot be estimated by inspection of the query and candidate.
- Functional distance is orthogonal to adaptation effort —or is it?

We hope not. Hence:

- Define measures of functional distance (that can be derived and compared by inspection).
- Correlate them statistically with actual adaptation effort.
- Select a measure as a good predictor of adaptation effort.

Premises:

- Measures of Distance are not numeric —do they have to be?
- They take values in a partially ordered set.
- They have well-defined mathematical formulas.
- Deriving and comparing functional distances amounts to proving theorems of first order logic.

2 Background: Lattice of Specifications

2.1 Refinement Ordering

Refinement Ordering:

$$R \sqsupseteq R' \Leftrightarrow R'L \subseteq RL \wedge R'L \cap R \subseteq R'.$$

Interpretation:

R subsumes R'.

Ordering properties: partial ordering.

2.2 Refinement Lattice

Any two relations R and R' have a meet (greatest lower bound), defined by

$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

Any two relations that satisfy $RL \cap R'L = (R \cap R')L$ have a join (least upper bound), defined by

$$R \sqcup R' = R \cap \overline{R'L} \cup R' \cap \overline{RL} \cup R \cap R'.$$

3 Measures of Functional Distance

We are given a library of software assets, say L, and a query K. Trying to characterize assets C that are optimal (with respect to some measure of distance) for query K.

3.1 Functional Consensus

Definition.

$$fc(K, C) = K \sqcap C.$$

Interpretation.

The functional consensus of K and C is the requirements information that is common to K and C.

Usage.

Retrieve all the components C of the library that maximize the measure of functional consensus with K, $fc(K, C)$.

Rationale.

Maximize the requirements of K that are covered by C.

3.2 Refinement Difference

Definition.

If $A \sqsubseteq B$, then the refinement difference between A and B is the smallest X such that

$$B \ominus A = A \sqcup X \sqsupseteq B.$$

We find

$$B \ominus A = B \cap \overline{AL} \cup (A \cap \overline{B})L \cap (B \cup \overline{A}).$$

Interpretation.

The refinement difference between A and B is the smallest amount of functional information that needs to be added to A to achieve or exceed B.

Usage.

Given query K, retrieve components C that minimize the quantity

$$rd(K,C) = K \ominus (K \sqcap C).$$

Rationale.

Minimize the requirements of K that left unfulfilled by C, as these measure how much must be added to C to make it satisfy (equal or exceed) K.

3.3 Functional Excess

Definition.

$$fe(K, C) = C \ominus (K \sqcap C).$$

Interpretation.

The functional excess of K and C represents the functional features of C that are irrelevant to K.

Usage.

Retrieve all the assets C of L that minimize the functional excess of K and C.

Rationale.

Minimize irrelevant features of C, that may get in the way of the adaptation effort (ref: program understanding).

3.4 Refinement Distance

Definition.

$$fd(K,C) = fe(K,C) \sqcup rd(K,C).$$

Interpretation.

Discrinimating information between K and C.

Usage.

Retrieve all the components C of the library that minimize the refinement distance between K and C.

Rationale.

Minimize required functional features of K and distracting functional features of C.

Mathematical properties:

- $rd(A, B) \sqsupseteq \emptyset$.
- $fd(A, B) = \emptyset \Leftrightarrow A = B$.
- $rd(A, B) = rd(B, A)$.
- $rd(A, B) \sqcup rd(B, C) \sqsupseteq rd(A, C)$.

3.5 Refinement Ratio

Definition.

$$fr(K,C) = \begin{pmatrix} rd(K,C) \\ fc(K,C) \end{pmatrix}.$$

Interpretation.

tems from those of refinement distance and functional consensus.

Usage.

Retrieve all the components C of the library that minimize the numerator and maximize the denominator.

Rationale.

Stems from those of the numerator and the denominator.

3.6 Functional Tangent

Definition.

$$ft(K,C) = \begin{pmatrix} fe(K,C) \\ fc(K,C) \end{pmatrix}.$$

Interpretation.

Stems from those of refinement distance and functional consensus.

Usage.

Retrieve all the components C of the library that minimize the numerator and maximize the denominator.

Rationale.

Stems from those of the numerator and the denominator.

4 Experimentation: Establishing Correlations

Library of 12 components. Sample of 14 queries. Six measures of distance. Estimation of Adaptation Effort.

4.1 Functional Distance

For each measure of distance, for each query K, observe how components are ranked with respect to their proximity to K.

Samples...

4.2 Adaptation Effort

Component	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}
Lexical analyzer	VH	VH	VH	VL	VH	VH	VL	VH	VL	VL	VH	VH
Syntactic analyzer	VH	VH	VH	VL	VH	VH	VL	VH	VL	VL	VH	VH
Semantic analyzer	H	H	H	N	H	H	VL	H	VL	VL	VL	VL
Global optimizer	N	N	N	L	N	N	VL	VL	VL	VH	VL	VL
Code generator	N	H	H	VL	H	H	H	VL	VL	VL	VL	VL
Peephole optimizer	VL	VL	VH	VL	VH	VH	VL	VL	VH	VL	VH	VH
Extra Code	VL	VL	VL	N	N	VH	VL	VL	VL	VL	N	VH

Defines Partial ordering; represented by a graph. Samples..

5 Results of the Experiment

Averaging behaviour of measures of distance over the set of 14 queries, with respect to the following criteria:

- Precision, judged against development effort.
- Recall, judged against development effort.
- Graph similarity, measured by ratio of common arcs.

5.1 Precision and Recall

Samples...

Summary:

Measure of Distance	Precision Avg (S.D.)	Recall Avg (S.D.)
Fun. Cons. Ref. Diff.	0.6292 (0.2992)	0.7611 (0.3852)
Ref. Dist. Ref. Ratio	0.3858 (0.2879)	0.4923 (0.3568)
Fun. Excess	0.4219 (0.2654)	0.7067 (0.3573)
Fun. Tang.	0.5658 (0.3561)	0.7917 (0.3965)

5.2 Measures of Similarity

Samples..

Specification	Similarity
K_1	0.396
K_2	0.410
K_3	0.333
K_4	0.326
K_5	0.438
K_6	0.431
K_7	0.438
K_8	0.278
K_{11}	0.368
K_{12}	0.563
K_{13}	0.389
K_{14}	0.410
Average	0.398
St. Dev	0.068

Averages of Similarity Measures.

Measures of Distance	Average Similarity	Standard Deviation
Fun. Cons. Ref. Diff	0.335	0.119
Ref. Dist. Ref. Ratio	0.461	0.162
Fun. Exc.	0.398	0.068
Fun. Tang.	0.367	0.159

5.3 Overall Ranking

Summary:

- Refinement difference and functional consensus rank first.
- Functional tangent is a close second.
- Functional Excess ranks third on all counts.
- Refinement Distance and Refinement Ratio rank fourth on all counts.

Observations:

- Functional Consensus is the best, and the easiest to compute.
- Refinement Ratio ranks worst, and hardest to compute.
- Refinement Distance, which is mathematically most elegant, ties for last with refinement ratio.
- Poor performance of functional excess is disappointing, given the propaganda about understanding costs.

6 Conclusion

Objective:

- Define measures of functional distance to predict adaptation effort.
- Correlate them to actual adaptation effort to assess their prediction capability.

Observations:

- Some measures perform consistently better than others.
- Simplest and easiest measures are best; mathematically elegant measures are worst.
- Measures that perform best in ppredicting adaptation effort and comparing candidates cannot possibly be used for making adaptation/new development decisions.
- Only an experiment; requires more validation.

Prospects:

- More experiments; very time consuming, as requires extensive theorem proving.
- Does the same measure apply to both adaptation decisions or do we need a separate measure for each?
- Do we need separate experiments?

Page intentionally left blank

Page intentionally left blank

OBJECT ORIENTED MODELING FOCUSED ON A LINGUISTIC APPROACH

N. Juristo, A.M. Moreno
Facultad de Informática - Universidad Politécnica de Madrid
Campus de Montegancedo s/n, 28660
Boadilla del Monte, Madrid
SPAIN
natalia@fi.upm.es, ammoreno@fi.upm.es

One of the main limitations attributed to Object Orientation by software engineers is the immaturity of the Object-Oriented analysis process. This article aims to propose an approach to formalize this process. This method is based on the use of linguistic information from informal specifications. This information is composed of words which, in turn, denote elements of an OO modeling, such as classes, properties, etc. These words have a particular meaning, and their use in the modeling is usually related with that meaning. So, the objective is to analyze this information from the semantic and syntactic viewpoint and extract, by means of a formal procedure, the components of an OO system. This paper presents briefly the proposed approach and is focused on the results of its application by a set of students of our university.

1. INTRODUCTION

The process of analyzing requirements is of essential importance in software development. The success or failure of a software system can be said to largely depend on the quality of this activity. A formal and disciplined process is therefore necessary for requirements analysis. However the situation is far away of this aim (Faulk,1997). We can find disagreement on terminology, on the approach and on the activities in the different methods; and, in the other hand, current methods do not usually provide formal, justified, complete and correct guidelines for identifying components of a problem that need to be represented in conceptual models.

The immaturity of the requirements process is particularly apparent with OO conceptual modeling, because, it is in its infancy. This insufficiency, and the need to remedy it has already been stressed by several authors such us (Iivari, 1995; Basili, 1996; Wang, 1997; Northop, 1997). They all emphasize the fact that there are no rigorous criteria for identifying components of OO conceptual models. They also claim that OO analysis cannot be effectively performed and its immaturity is slowing down the adoption of OO.

We have developed an approach that seeks to formalize the analysis process so as to create conceptual models in a rigorous and precise manner. We have focused on OO modeling, because this is one of the least mature areas.

The proposed approach is based on examining information most likely to be available at the start of development, i.e., sentences in natural language that describe characteristics of the problem to be solved. This description is composed of words and these words can serve as elements of a conceptual model. We will focus on formal definition of relations between words or linguistic structures and elements of modeling or conceptual structures.

Our approach consists of two different activities: conceptual modeling formalization and OO model creation. In this paper we present briefly, the bases of the formalization and the steps to be followed during the OO model construction, going on to describe the main results obtained after the application of this approach.

2. FORMALIZATION

The formalization provides defined rules to identify key elements of conceptual models by defining relations between a subset of structures from linguistic world and a subset of structures from conceptual world. Linguistic world is potentially unlimited, which led us to work with a subset, called utility language. Linguistic structures that compose this subset are referred to as *linguistic patterns*. Regarding to conceptual world, it is formed by any conceptual models that represent a problem and its solution. In this case, we are going to work with two OO conceptual models, the Object Model (OM), which will represent that static structure of the problem, and the Behavior Model (BM), which will represent its dynamic aspect. Conceptual structures of these models constitute what are called *conceptual patterns*.

Figure 1 shows the reasoning followed for the definition of a formal correspondence between linguistic (L) and conceptual (C) patterns. It is based on the equivalence between their mathematical representations. In particular, between the equivalence of the logical representation of linguistic patterns (PL) and the set theory representation of conceptual patterns (ST). More details about this formalization can be found in (Moreno, 97b).

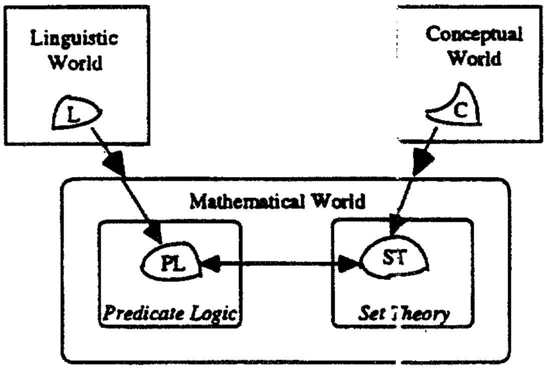


Figure 1. Relation between the Linguistic ar.d the Conceptual Worlds

The result of the formalization is the correspondence shown in Table 1, where (1) names of classes are equivalent to the nucleus of the noun structure of noun groups or complements, and (2) names of relations are equivalent to the verb in third person singular; and (3) subordinate_1, ..., subordinate_n represent any simple clause in a subordinated clause.

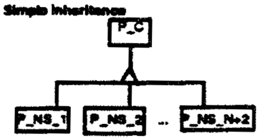
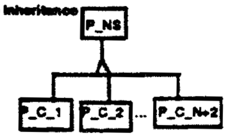
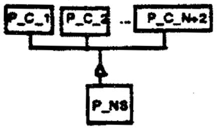
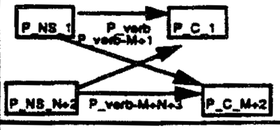
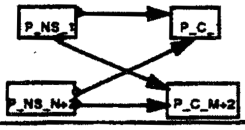
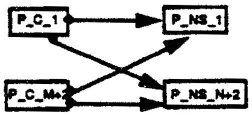
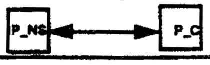
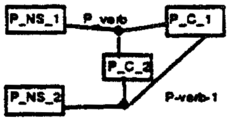
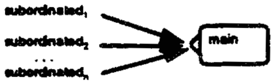
LINGUISTIC PATTERN	CONCEPTUAL PATTERN
l_1 Classification *$l_{1.1}$* Bottom-up classification Nominal Group [[, Nominal Group] ,* and|or Nominal Group] — bottom-up_clasification_verb — Complement	c_1 Simple Inheritance 
$l_{1.2}$ Top-down classification Nominal Group — top-down_clasification_verb — Complement [,Complement]* or Complement	c_1 Simple Inheritance 
$l_{1.3}$ Multiple classification Nominal Group — multiple_clasification_verb — Complement [,Complement]* and Complement	c_2 Multiple Inheritance 
l_2 Complement enumeration Nominal Group [[, Nominal Group] ,* and |or Nominal Group] — general_verb — Complement [[, Complement],* and| or Complement]	c_3 Binary Association  P_NS_1 P_verb P_verb-M+1 P_C_1 P_NS_N+2 P_verb-M+N+3 P_C_M+2
l_3 Composition *$l_{3.1}$* Component composition Nominal Group [[, Nominal Group] ,* and| or Nominal Group] — component_composition_verb — Complement [[, Complement],* and|or Complement]	c_4 Agregation 
$l_{3.2}$ Content composition Nominal Group [[, Nominal Group] ,* and| or Nominal Group] — container_composition_verb — Complement [[, Complement],* and| or Complement]	c_4 Agregation 
l_4 Identification Nominal Group — identifies — Complement	c_5 Identification Association 
l_5 Adjacent complements Nominal Group [[, Nominal Group] ,* y|o Nominal Group] — general_verb — Complement [Complement] ,*	c_6 N-ary association 
l_6 Condition *If only if* — subordinated — *then* — main	c_7 Behaviour  subordinated_1 subordinated_2 ... subordinated_n main

Table 1. Correspondence between Linguistic and Conceptual Patterns

3. OO MODEL CREATION

OO model creation employs the results of the formalization and guides analysts in building conceptual models. This activity is achieved by means of the steps shown in Figure 2, which are detailed in (Moreno, 97a). The first five steps prepare the problem description for application of the formalization output, which will be used during steps 6 and 7. The other tasks in these steps as well as steps 8 and 9 combine conceptual patterns to form the OM and the BM.

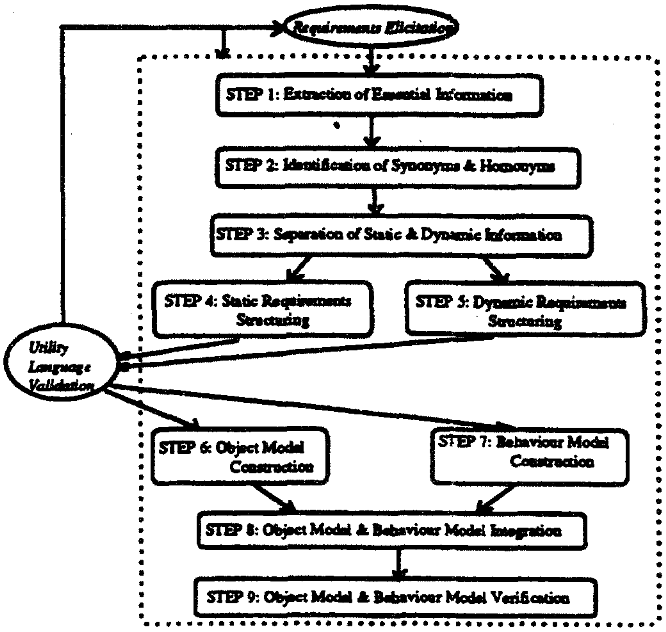

Requirements Elicitation
STEP 1: Extraction of Essential Information
STEP 2: Identification of Synonyms & Homonyms
STEP 3: Separation of Static & Dynamic Information
STEP 4: Static Requirements Structuring
STEP 5: Dynamic Requirements Structuring
Utility Language Validation
STEP 6: Object Model Construction
STEP 7: Behaviour Model Construction
STEP 8: Object Model & Behaviour Model Integration
STEP 9: Object Model & Behaviour Model Verification

Figure 2. OOA Method Steps

Our approach seeks to give precedence to one of the empirically existent conceptual representations that can model a given problem. The goal is to ensure that the conceptual models adequately represent the studied problem and its solution. The fitness of these models will be formally justified by employing the results supplied by the formalization during the construction process, that is, during the OO method application.

4. RESULTS OF THE APPROACH APPLICATION

With a view to refining our approach, we have worked with a group of final-year degree students at the School of Computer Science in the Politechnical University of Madrid. Some of them were taught the proposed approach and others the standard OMT (Rumbaugh, 1991) approach. In this manner, we sought to achieve a second objective: compare how good the two methods were with respect to obtaining conceptual models by people with no experience in OO.

In order to get representative results, that is, reliability of 95% and standard deviation no higher than 2.28, it can be considered that a sample of size 25 from an homogeneous poblation of about 100 students would be adequate. Also note that both approaches were taught strictly following the rules provided by their authors.

Detailed results of this application are documented in (Moreno, 97). The most interesting conclusions are discussed below.

- Analysts working with our approach were thought about the problem more, instead of directly setting out creating models, a common mistake made by inexpert analysts. This implies better study and understanding of the problem under analysis, a task that is essential in performing a good analysis.
- Analysts working with our approach expended more time before conceptual modeling. This time was used in creating the utility language, that is the representation of the problem following the linguistic patterns. Analysts working with OMT expended significant time discussing which elements should form part of conceptual models, which led to incorrect models in some cases. This can be attributed to the absence of rigorous criteria for identifying elements of conceptual models. Using our approach, 65% of the time was spent transforming to the utility languages, while 35% was spent constructing conceptual models. Using OMT, 85% was spent constructing conceptual models, and 15% was spent understanding the problem before focusing on conceptual models.
- Our approach avoid some kinds of incorrect modeling constructions, due to the encapsulation of OO concepts, which may not be familiar with, nor completely understood by inexpert analysts, under the concepts of the language, which are known and generally used by analysts. In that sense analysts working on OMT have developed incorrect conceptual structures due to the misunderstanding of some OO concepts. For example, as can be seen in Figure 3, which represents the static part of a video club system in OMT, one of the most common errors is the incorrect use of the inheritance concept, where a copy of a movie is drawn as subclass of movie. This mistake did not occur using our approach, as in order to get that conceptual pattern, there should be a linguistic pattern saying for example "a copy of a movie is a kind of movie" which is not correct from a semantic point of view.

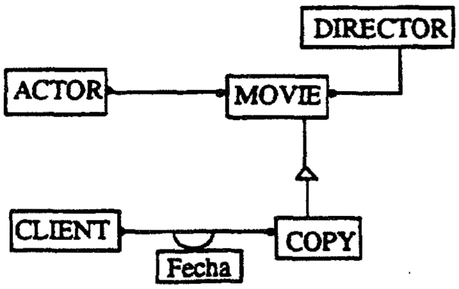


Figure 3. OMT Object Model

- The application of our approach makes the validation process easier for users. The reason is that part of the validation can be done before building the conceptual models, during the Validation Utility Language process, seen in Figure 2. This utility language is, as it has been said, natural language which the user is familiar with. In the other hand, validation of conceptual models can be more difficult for users as usually, they will not be familiar with specific notations that depend on software development methods and technical aspects. The proper conceptual models will be achieved by means of the correct application of the correspondence between linguistic and conceptual patterns, shown in Table 1.
- Conceptual modeling using our approach is more repeatable. For example in case of dynamic representation of the system, all conceptual models are very similar to the one of Figure 4. On the other conceptual models got in OMT are very different, above all in the description of the dynamic part of the system using state transition diagrams. This is due to the application of the correspondence between linguistic and conceptual patterns. We consider this characteristic very important, as it is one of the prerequisites for measuring the process, and use these measures to improve it.

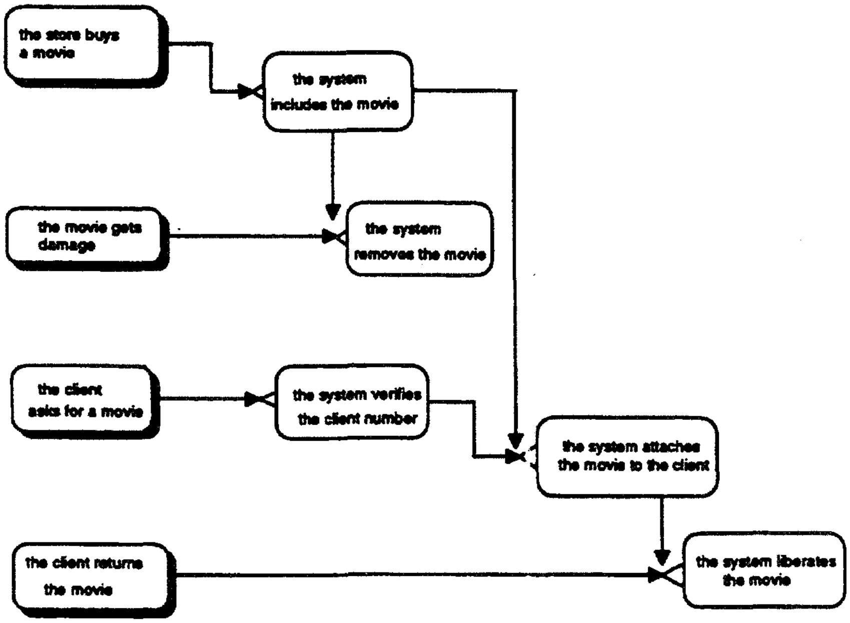

the store buys a movie
the system includes the movie
the movie gets damage
the system removes the movie
the client asks for a movie
the system verifies the client number
the system attaches the movie to the client
the client returns the movie
the system liberates the movie

Figure 4. Behavior Model with the proposed approach

5. CONCLUSIONS

Our work can provide a formal approach in order to get conceptual models that represent the problem and its solution. The use of a linguistic approach has led us in the obtention of adequate conceptual models, from all models that could empirically represent a given problem. The choice of these models is based on the use of the mathematical world as catalyst for converting linguistic world into conceptual world.

One of the most interesting results of this approximation is that its application is easier than the application of any other current method. The reason is the use of linguistic knowledge to get conceptual representations. This can be really useful for non expert analysts who are not completely familiar with OO concepts. They would have to apply the steps seen in section 3 to get the conceptual models, it wouldn't be necessary to know the details of the formalization, only the results, that is, the correspondence between linguistic and conceptual patterns.

The same correspondence can be applied in order to obtain natural language from the conceptual models. That is what is known as paraphrasing, and is used to make easier the validation process. From conceptual models the analyst can get the description in natural language of these models, and the user can validate this description instead of validating conceptual models.

Finally we can say that our approach would be easily automatizable.. In fact we are planning to develop a tool to automate the OOA model construction, that is, a tool to guide analysts in construction of conceptual models of a problem. This involves inputting the results of the formalization into the tool. It would really be useful during Steps 6, 7, 8 and 9 of the method, which is when the results of formalization are applied and when conceptual models of the problem are built. It should also provide support for the earlier steps of the method, that is, Steps 1 to 5. These steps require significant intervention on the part of analysts, for which the method provides the necessary criteria. However, they are not automatic processes.

6. BIBLIOGRAPHY

Basili, V.R., Briand, L.C., Malo W.L., *How Reuse Influences Productivity in Object Oriented Systems*. **Communications of the ACM**, 39 (10), 104-116 (1996).

Faulk, S.R., *Software Requirements: A Tutorial.* In **Software Engineering**, 82-101, IEEE Computer Society Press, Los Alamitos (1997).

Iivari, J., *Object-Orientation as Structural, Functional and Behavioral Modeling: A Comparison of Six Methods for Object-Oriented Analysis*. **Information and Software Technology**, 38, 155-163 (1996).

Moreno, A.M. **A Conceptual Modeling Formal Method for Software Systems**. PhD Thesis, Universidad Politécnica de Madrid, Madrid (1997).

Moreno, A.M. *Object Oriented Analysis from Textual Specifications.* **Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering**, Madrid, June 17-20 (1997).

Moreno, A.M.; van de Riet, R.P. *Justification of the Equivalence between Linguistic and Conceptual Patterns for the Object Model.* **Proceedings of the International Workshop on Applications of Natural Language to Information Systems**, Vancouver, June 27-29, (1997).

Northrop, M., *Object-Oriented Development.* In **Software Engineering**, 148-159. IEEE Computer Society Press, Los Alamitos (1997).

Rumbaugh, J., et al., **Object-Oriented Modeling Technique**. Prentice-Hall, New Jersey, 1991.

Wang, S., *A Synthesis of Natural Language, Semantic Network and Objects for Business Process Modeling*. **Canadian Journal of Administrative Sciences**, 14(1), 79-92 (March 1997).

OBJECT-ORIENTED MODELING FOCUSED ON A LINGUISTIC APPROACH

N. Juristo, A. M. Moreno
Facultad de Informática - Universidad Politécnica de Madrid
SPAIN

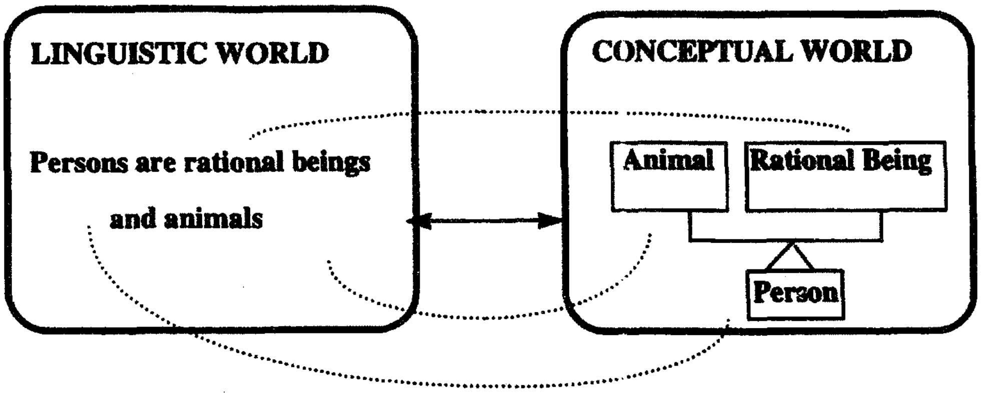
PROBLEM ANALYSIS

"Current methods do not provide formal, justified, complete and correct rules for indentifying components of a problem to be represented in conceptual models"

SPECIAL GAP:

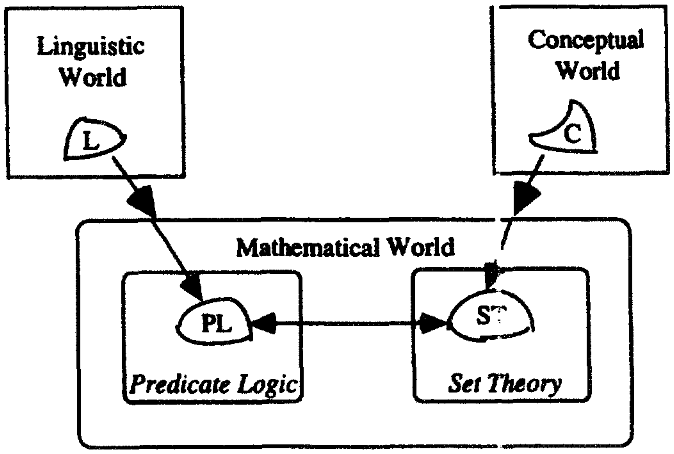
OBJECT ORIENTED ANALYSIS

OBJECT ORIENTED ANALYSIS

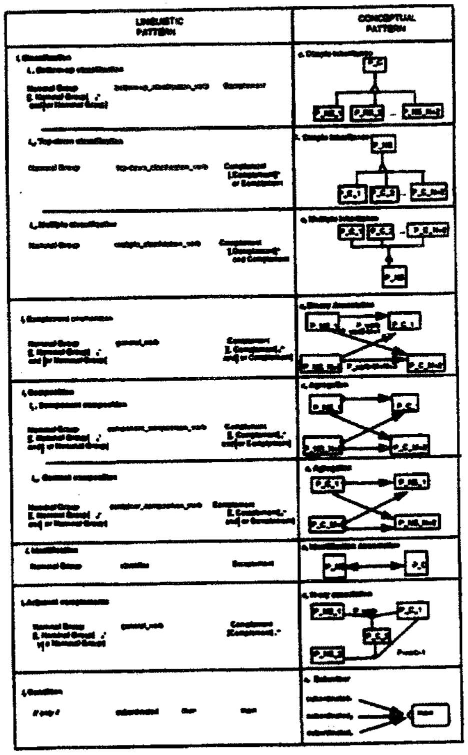


- Conceptual Modeling Formalization
- OO Model Creation

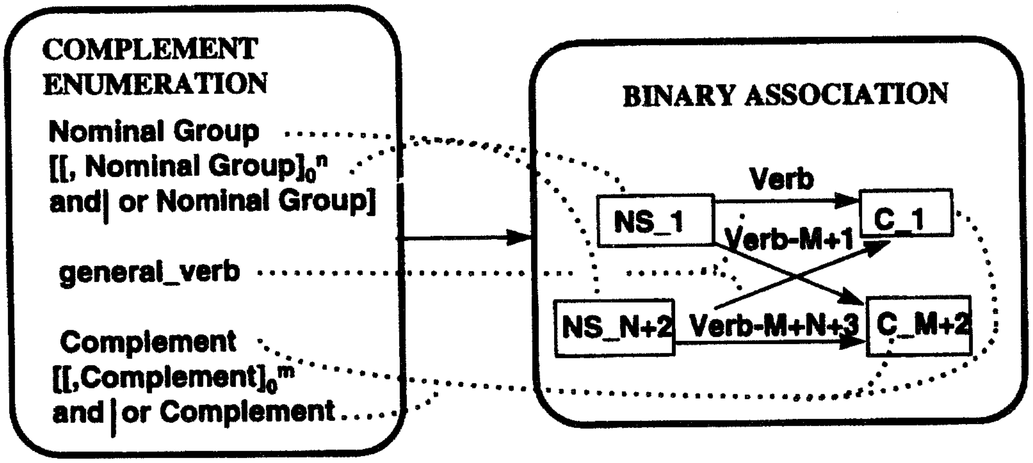
CONCEPTUAL MODELING FORMALIZATION (i/iii)



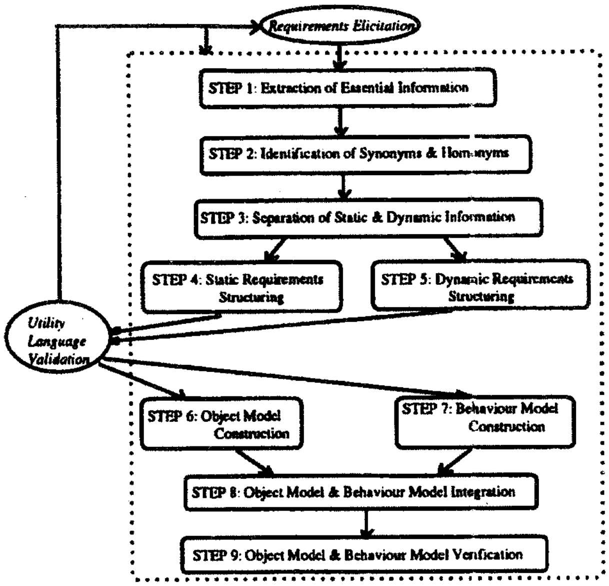
CONCEPTUAL MODELING FORMALIZATION (ii/iii)


LINGUISTIC PATTERN
CONCEPTUAL PATTERN

CONCEPTUAL MODELING FORMALIZATION (iii/iii)



OBJECT ORIENTED MODEL CREATION



Requirements Elicitation

STEP 1: Extraction of Essential Information

STEP 2: Identification of Synonyms & Hom-nyms

STEP 3: Separation of Static & Dynamic Information

STEP 4: Static Requirements Structuring

STEP 5: Dynamic Requirements Structuring

Utility Language Validation

STEP 6: Object Model Construction

STEP 7: Behaviour Model Construction

STEP 8: Object Model & Behaviour Model Integration

STEP 9: Object Model & Behaviour Model Verification

RESULTS OF THE APPROACH APPLICATION (i/iv)

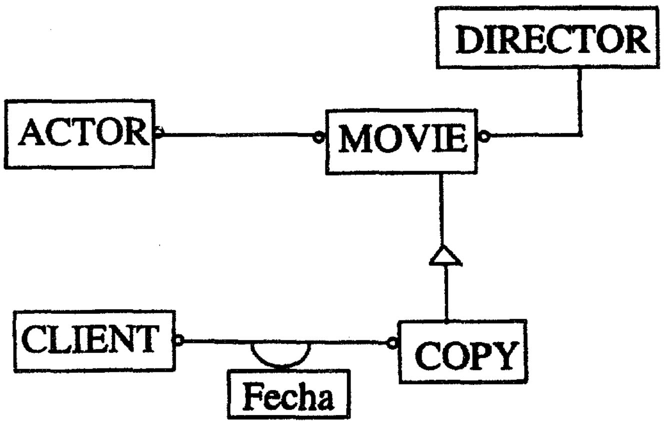
OUR APPROACH

- Analysts think about the problem
- Conceptual model construction time: 35%
- OO concepts encapsulated in linguistic structures

OMT

- Analysts directly construct conceptual models
- Conceptual model construction time: 85%
- Pure OO concepts

RESULTS OF THE APPROACH APPLICATION (ii/iv)



RESULTS OF THE APPROACH APPLICATION (iii/iv)

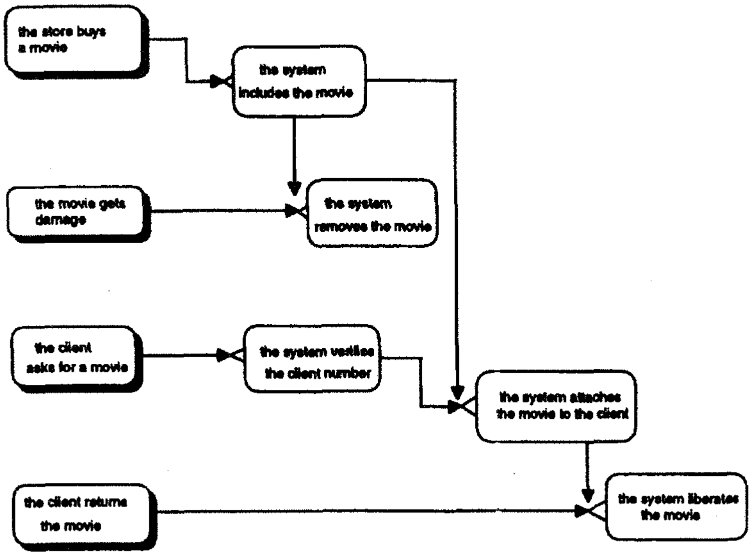
OUR APPROACH

- **Validation of natural language**
- **Conceptual modeling repeatable**

OMT

- **Validation of conceptual models**
- **Very differents conceptual models**

RESULTS OF THE APPROACH APPLICATION (iv/iv)



the store buys a movie

the system includes the movie

the movie gets damage

the system removes the movie

the client asks for a movie

the system verifies the client number

the system attaches the movie to the client

the client returns the movie

the system liberates the movie

CONCLUSIONS

- Formal and systematic analysis approach
- Easy application by inexpert analysts
- Paraphrasing for validation purpose
- Automatization possibilities

Case Study of an Object-Oriented System: EOSDIS

Jeanne Behnke and Susan Sekira

Codes 505 and 514

360876

Mission to Planet Earth (MTPE) is a long-term NASA research mission to study the processes leading to global climate change. The Earth Observing System (EOS) is a NASA campaign of satellite observatories that are a major component of MTPE. The EOS Data and Information System (EOSDIS) is another component of MTPE that will provide the Earth science community with easy, affordable, and reliable access to Earth science data. EOSDIS is a distributed system, with major facilities at six Distributed Active Archive Centers (DAACs) located throughout the United States. The EOSDIS software architecture is being designed to receive, process, and archive several terabytes of science data on a daily basis. Thousands of science users and perhaps several hundred thousands of non-science users are expected to access the system. While there are many segments in EOSDIS (e.g., flight operations, network) this case study discusses the development of the science data processing segment (SDPS). We briefly review the architecture of the system, the goals of the SDPS, and the development progress to date. This study highlights key software development challenges, experiences integrating COTS, and the difficulties of managing a complex system development effort.

The EOSDIS data collection begins at the ground systems. From the ground systems, the data is forwarded to the SDPS where the data is captured, processed, archived and distributed. The SDPS is being developed as a fully distributed and heterogeneous system using object-oriented (OO) development methodologies. The SDPS, the focus of this paper, is decomposed into seven subsystems:

1. INGEST subsystem - receives data from external and internal sources and submits them for archive
2. DATASERVER subsystem - archives and distributes data
3. PLANNING subsystem - develops plans for producing data products (level 0 to level 1)
4. DATA PROCESSING subsystem - manages, queues and executes processes for the generation of data products
5. CLIENT subsystem - allows users to access the services and data available in the system
6. INTEROPERABILITY subsystem - provides the software infrastructure for the communications between clients and servers in the system
7. DATA MANAGEMENT subsystem - supports the location, search, and access of data and services.

It is easier to appreciate the magnitude of the design problem, if you look at the scope of the data volumes to be processed. EOSDIS, at all DAACs, will have to support at least 260 different data products and sets of raw instrument data. These DAACs can expect at least 480 GB of raw instrument data and subsequent processing will create approximately 1.6 TB of data, all to be stored daily. At one of the largest DAACs, located at Goddard Space Flight Center, the permanent archive is expected to reach about 245 TB with more than 15 million files. This DAAC will expect to ingest around 574 GB per day (a throughput rate of 6.64 MB/s into the archive) and disseminate more than 368 GB/day (a throughput rate of 4.26 MB/s out of the archive). In addition to providing a comprehensive data retrieval and processing system, the SDPS is being tasked to provide a flexible, scaleable and reliable system. The architecture must be capable of supporting:

- new data types with minimal software modifications
- new data centers that will not require new code and software agreements
- standard interfaces (HDF-EOS) enabling coordinated data analysis
- data access from a wide variety of users (e.g., kindergarten teachers, as well as college professors)

- technological advances and the infusion of new COTS products and techniques (e.g., data mining)
- inevitable change and new requirements

To meet the challenge of the SDPS, the EOSDIS Core System (ECS) was designed (see Figure 1). It is an enormous development effort comprising 75 COTS packages, about 1 million lines of code and the efforts of approximately 220 developers. At this time, about 90% of the launch critical capabilities have been completed and are in the process of being tested. On August 28, 1997, a successful demonstration of this segment of the ECS was held. The demonstration was organized into scenarios to give reviewers the best overview of the developing system. Using an operational construct, the reviewers were shown how data from the Landsat-7 and AM-1 missions would be ingested, processed, archived and distributed. Although the August milestone was successfully met, the overall system is still far from completion. Since the onset of the development effort two years ago, the EOSDIS project has had to re-examine its software processes and make numerous changes along the way. This case study highlights the experiences and reasons for change.

Software Development Challenges:

With a software development effort of this magnitude, it is often easy to lose perspective. To complete such an endeavor from a developer standpoint, it is imperative that the requirements be fully understood, the development effort be properly estimated, that contingencies and changing requirements are planned for, and an attainable schedule is developed. To accomplish the task, the EOSDIS contractor chose several techniques to manage the effort. We focus on the design definition, the development methodology, choosing object-oriented metrics, and successful software practices.

The project continues to refine its software engineering practices for the best way of capturing and documenting the design. What originally seemed appropriate documentation at the detailed design stage is no longer viewed as crucial information to maintenance or test personnel. At the outset, the ECS was designed following the Object Modeling Technique (OMT) Methodology, which emphasizes a number of techniques and notations for analysis and design. Using a tool called Software-Through-Pictures (StP), the design documentation included event traces and state diagrams, which were then used by programmers for initial software development. While these can assist in refining the object models, later on in the development life cycle they tend to be less favored design vehicles. Customers and evaluators, in particular, found that there was insufficient detail in the class and method descriptions to understand the design. Also, they found that the event traces were not very useful. Many, both developers and customers, preferred to view functional diagrams and scenario definitions. Programmer familiarity and experience has played a key role in defining the documentation needs. Currently, the development staff uses Rational Rose, from Rational Software Corporation, for forward and reverse engineering. Discover, a software engineering tool from Software Emancipation Technology, was selected to provide software impact analysis and code comprehension. ABC++, a product developed under Lockheed-Martin, displays classes, objects, functions and attributes via an html browser.

System requirements were laid out to a level 3 specification under the contract. These level 3s are currently under configuration control through a CCB process. The level 3s were further broken down by the contractor, creating level 4 requirements. These level 4 details are in turn mapped not only to the subsystems and particular code drops, but also to the OO classes. This mapping has subsequently become quite a burden to maintain. It is difficult to keep up with changing requirements, understanding the heritage of the level 4 and volatile scheduling.

In the early days of the project, considerable time was spent on benchmarks and prototypes. This early evaluation of the components and system requirements was useful to determine how they might best be incorporated or eliminated. For example, several benchmarks of database management systems were undertaken at different times to determine which product would best support the ECS. The latest benchmark looked at spatial query processing of a 100GB database application, where clients were simulated connecting to the server in small and large groups. Another example of benchmarking was early examination of network attached storage (NAS). At the time, NAS was (and still is) very popular, but it was determined that the data transfer rate would be unable to support the high data rates of ECS. The user interface was prototyped a number of times in an effort to meet end user needs.

Another aspect of developing the design was to establish a number of Working Groups. These committees convened to discuss such subjects as:

- the metadata model and schema
- system access pattern by end users
- data characterization and size estimation

One of the best ways to understand the design has proven to be through workshops and scenario-based demonstrations. Workshops and demonstrations have been key tools for viewing ongoing development and EOSDIS concepts. This use of demonstrations provides early insight into potential architectural and programmatic deficiencies and permits users and testers of the system a chance to view the system prior to completion.

Early in the development effort, it became apparent that the *single delivery* approach would not make schedule. The project re-examined processes for potential problems. One area of concern was the choice of the Waterfall Software Development Methodology. Used traditionally for mainframe development, this methodology was not suited to the client/server nature of the system. Under this methodology, the concept was to complete an entire build and deliver it all at once to the Test team. Another facet of this approach was to implement an incremental track for development. Each subsystem was assigned to a specific track independent of each other. This approach, however, does not show end-to-end functionality during the development cycle. As a result, a new process for software development was adopted, similar to the Spiral Model. This new process, called the Evolutionary Software Build Methodology, outlines an iterative approach to software development whereby software functionality is built into the system in small steps. So the present approach is called *build a little, test a little* and the key focus is to demonstrate functionality. These demonstrations then buy some measure of customer confidence in completion of this large system. The development approach has mapped out several drops of the system. These drops focus sizable chunks of functionality that meet user and tester needs. The drops are assigned basically as:

- what is needed prior to launch (science software and integration and test)
- what is needed at launch (launch critical)
- what is needed 60 days after launch (launch critical + 60)
- and so on...

User needs were determined by DAACs, Instrument Teams and Test Teams. This method of deploying drops allows time for addressing performance issues in conjunction with development and integration. Mode management is used to test different aspects of the system independently and simultaneously.

Many software development practices have been established to guide the development process. Numerous program instructions have been written to provide software guidelines regarding coding standards, naming conventions, and configuration management. Software development files are also created by developers to maintain historical design information representing the current state of their software. Programmers have access to several tools for development like Builder Xcessory and S-Designer. At this time, ECS estimates that 250 source lines of code are produced by a programmer per month. Once a set of classes has been completed by the programmer, a code walkthrough is scheduled. The code is reviewed by peers and the lead programmer, as well as a quality assurance reviewer. Code issues are recorded and the programmer makes the necessary adjustments. Prior to unit test, Purify, from Rational Software Corp., is run on the code segments to test for memory leaks and execution errors. The next step is to promote the code to the appropriate build/merge segment within the ClearCase environment, a configuration management tool also from Rational. ECS has also instituted daily integration meetings to support the software development effort. At these meetings, the developers keep track of problems, note changes in the environment, and coordinate the integration effort. ECS tracks defects in the code through non-conformance reporting (NCR) mechanisms. An NCR can be filed by a developer, integrator or tester. DDTS (Distributed Defect Tracking System), yet another tool from Rational, is used to track NCRs. For planning purposes, ECS predicts that 3 NCRs will be found per 1000 source lines of integrated code and that it will take a programmer 20 hours to analyze an NCR *problem* and code and unit test the fix. One area which presents a high percentage of problems for ECS is the configuration of clients and servers. To manage the configuration parameters needed to correctly initialize the servers, ECS is developing a program called ECSAssist which will allow interactive configuration of the servers.

Another challenge to developing this system was selecting the appropriate methods by which to qualitatively and quantitatively measure software development. While Object-oriented analysis and design continues to gain popularity, there are still limited approaches for gathering OO software metrics. Similarly, there are few documented OO baselines for establishing productively rates and assessing quality. McCabe tools are now being used to measure reusability and maintainability (e.g., depth of inheritance trees and number of children). A random sample of 5% of the code showed a complexity factor of greater than 10. McCabe is also being used to examine the number of classes and number of methods. Unfortunately, several problems arise when evaluating the metrics. It is often difficult to isolate the COTS software from the custom software. For this project, there also seems to be a significant amount of code that is either old/obsolete or not promoted to integration, which tends to skew the metrics. As ECS polishes the completed software, we anticipate more accurate metrics from the available tools.

Integrating COTS:

It would be difficult to find a project at NASA that tries to integrate so many different 'cutting edge' hardware and software packages. It is often an easy decision to select a particular package based on its reported performance and determine that it is a good solution for the entire project. The decision to use so many products and platforms was driven by the desire to build a system to operate at least twenty years and to have the flexibility to grow and change with those years. This is not a decision unique to the EOSDIS project. This system uses approximately 75 off-the-shelf (OTS) packages from commercial and government sources. Principal packages (that drive the design) include:

Sybase Relational DBMS/SQS	dbms and spatial query system
AMASS	file storage management system for robotic storage devices
Autosys	scheduling software for the processing system
Tivoli	system management tools
HP Openview	graphical tool for system management
RogueWave	libraries used to map components to objects
DCE and OODCE	distributed computing environment
ClearCase	CM tool to manage completion of different builds
Remedy	trouble-ticketing software used across project

ECS has chosen to implement the design on 3 base platforms: SGI Challenge Servers, SUN Enterprise Servers and HP J210 and K420. The mass storage systems selected for ECS are also important design drivers. For the fast archive, EMASS AML/2 robotic units were chosen along with HP magneto-optical drives. The EMASS units will house up to 4576 5.25" optical disks for a maximum capacity of 12 TB. StorageTek STK Powderhorns form the base archive robotic devices, each supporting 16 SONY D3 drives.

The D3 tape media holds 50 GB for approximately 250 TB in a storage unit. Each mass storage unit is scaleable as new tape devices and media are developed.

Unfortunately, many development problems can be attributed to COTS integration. A significant learning curve is associated with each COTS package. Some of the technical problems encountered include proper tuning of the products per platform, maintenance of baseline products and a mismatch of product availability to when it is needed. For example, ECS seeks to maximize the use of particular hardware systems for optimal performance. SGI was chosen as a target platform because it has an enhanced I/O bus structure that will allow ECS to provide accelerated access to the data in the storage archive. However, SGI is not a first string platform for SYBASE and there are delays in getting qualified SYBASE products for the SGI. ECS has also chosen to implement Distributed Computing Environment (DCE) structures for inter-process communication. In fact, since the system has been designed using an OO methodology, OODCE has been implemented in parts of the design. SUN has implemented DCE within the operating system, where SGI chooses to view it as a layered product. Consquently , the support for DCE on SGI platforms is secondary to that company. This project has to continuously deal with products that are not as mature as required and have problems in a multi-platform environment. Use of HP, SUN and SGI as the basic platforms for the architecture has proved difficult, despite a reliance on hardware and software standards. These comments do not reflect on the OO design of the code but are probably indicative of any project of this size and complexity. To combat these problems, ECS brings in consultants to immediately attack such problems. Another solution is to continuously meet with COTS vendors to push for solutions. In a few cases, NASA has paid to improve a product to meet our needs.

Management of a Complex Development Effort:

There are a few key issues that bear particularly upon the management of the software development. These issues encompass the characteristics of the EOSDIS contract, the staffing profile, and schedule management. Hughes Information Technology Systems holds the prime contract for EOSDIS development, however it employs a number of sub-contractors who specialize in a variety of areas. For example, EDS is contracted to manage hardware and software procurements, while NYMA is responsible for system verification. Just to give perspective on the staff size of the development effort, approximately 220 developers are employed by ECS. Almost all of the contractors are located in Landover, MD in close proximity to Goddard Space Flight Center where the NASA project managing the EOS effort is located. The development staff is organized hierarchically; programmers have been assigned to subsystems and subsystems are managed by a lead programmer. A new change has been to co-locate government personnel at the Landover site, which allow issues to be handled immediately and reduces the number of formal meetings.

Since the EOSDIS contract was awarded, the contractor for SDPS has been faced with a number of critical issues. A key issue has been the staffing profile. The highly-specialized skills required to program objects in C++ are much sought after in the Washington metropolitan area. Hiring and attrition have been problems for the contractor. For example, financial compensation has not proved to be a good solution to attrition. One method to compensate for staff turnover has been to develop in-house training programs.

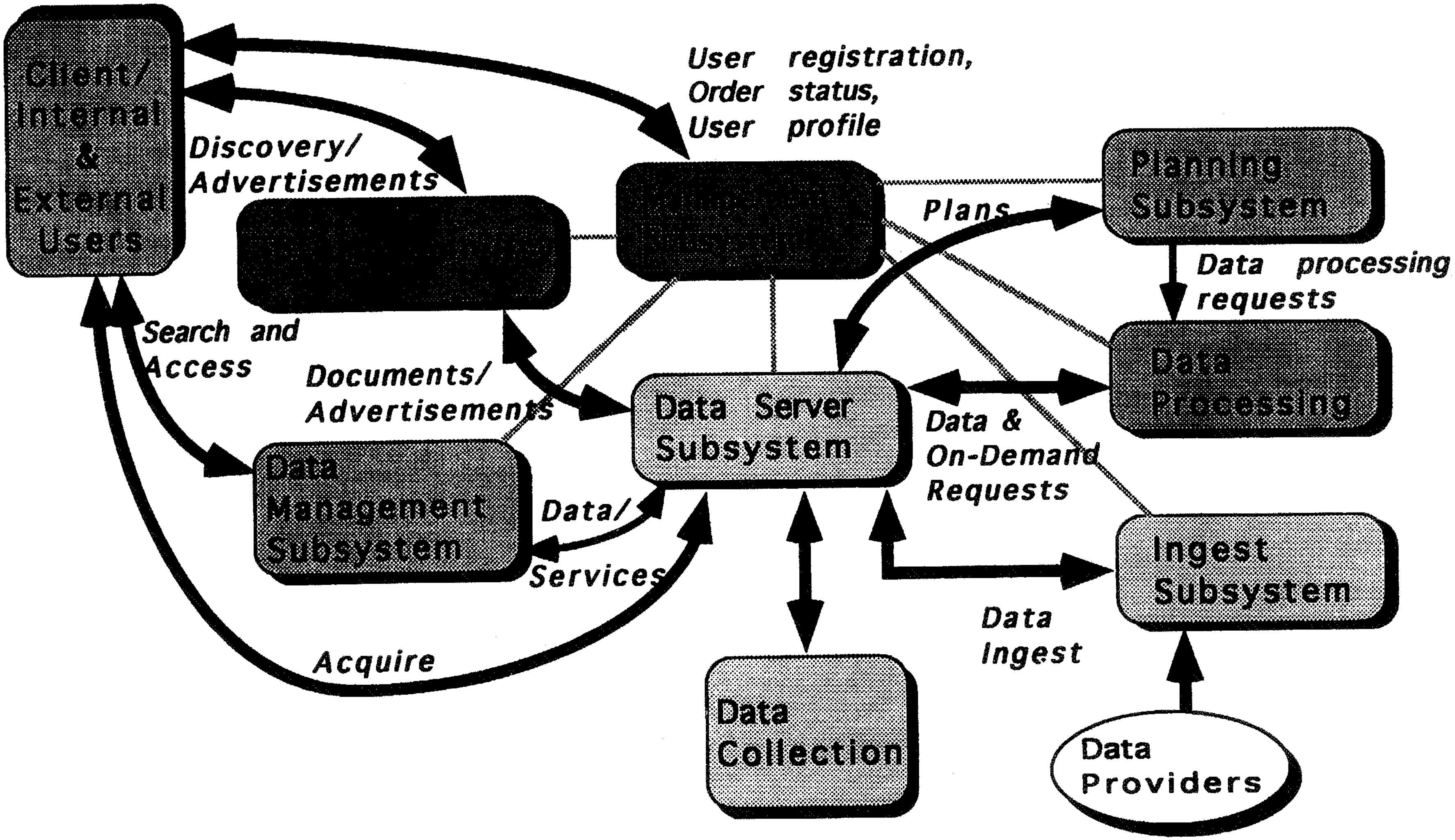
Another critical issue has been resource and software development scheduling. The schedule for software development has been replanned three times in the last two years. Part of the scheduling problem has been the frequency with which unplanned work has entered into the software development effort. Unplanned work is possibly a symptom of an object-oriented software development effort. Unanticipated problems with COTS and hardware platforms have also contributed to delays meeting schedule. These problems are exacerbated by the number of COTS packages used by the project. The iterative approach to software development has led ECS to schedule parallel development efforts. Despite the careful attempts at scheduling, subsystems found themselves working parallel branches of development at the same time, which led to very long work hours. The time alloted to subsystem integration at the outset did not reflect the actual time it took. A year ago, it took several days to complete a merge that now can be performed in hours. However, this integration time was not reflected in early schedules.

Several steps have been taken to improve our scheduling process. A new schedule has been drafted that schedules completion of the launch critical elements of SDPS by the launch of AM-1 in June 1998. To ensure that the launch date can be met, ECS is using a new scheduling tool, Primavera Project Planner from Primavera Systems, Inc. The new schedules factor in everything, from vacation time of the staff to time for developing, testing and deploying. Extra time is also allocated for integration, bug fixes, training and documentation. Since Primavera allows full resource management, ECS is better able to generate a more accurate schedule. While there are many aspects to effectively managing the ECS project, schedule and staff are the two most critical drivers in the success of this project.

Conclusion:

EOSDIS is an ambitious program and a keystone in the Mission To Planet Earth. Many things can be learned from the development of this system. This paper focused on some of the key aspects of system development and software management, COTS integration and program management. However, there are many other aspects to the development of this system that were not even mentioned. Perhaps other projects can see parallels to the EOSDIS adventure. We encourage further discussion on these topics.

ECS Context Diagram



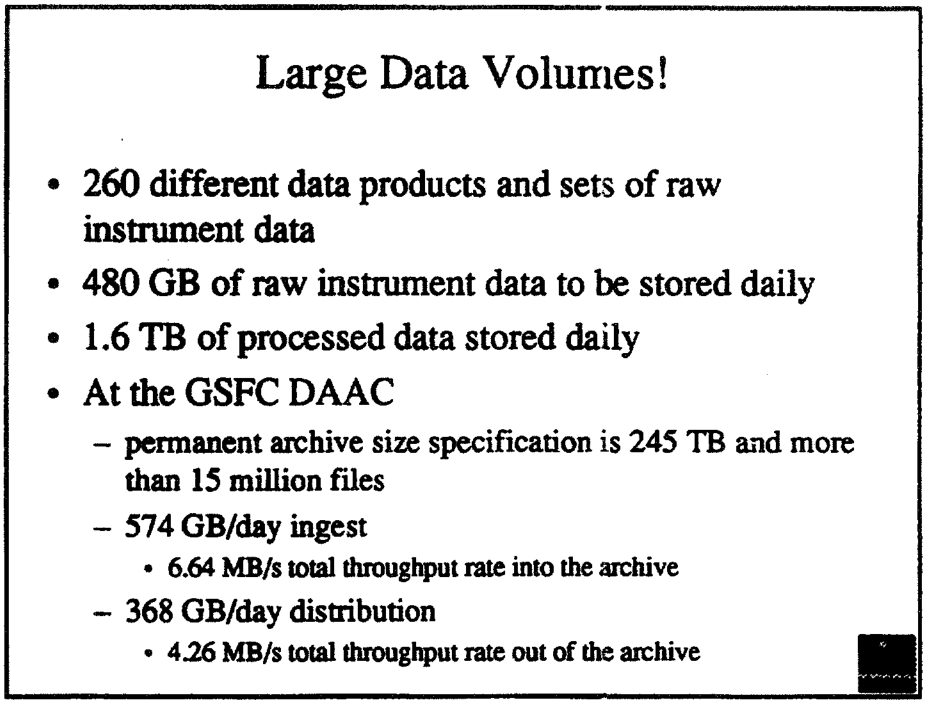
A Case Study of an Object-Oriented System: EOSDIS

Jeanne Behnke
Sue Sekira
Earth Science Data Information Systems

Software Engineering Workshop
December 4, 1997

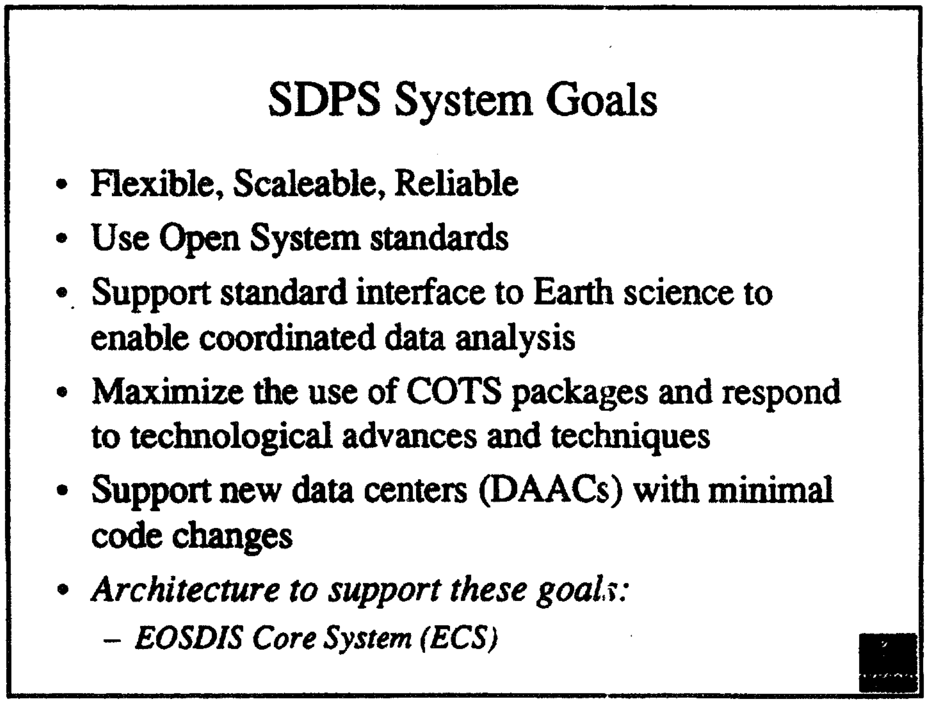
EOSDIS Concept

- EOSDIS is a distributed system
  - 6 major facilities across the US
    - called Distributed Active Archive Centers (DAACs)
- Software architecture is designed to receive, process, archive and distribute several terabytes of science data on a daily basis
- User community consists of several thousands of science and non-science users



Large Data Volumes!

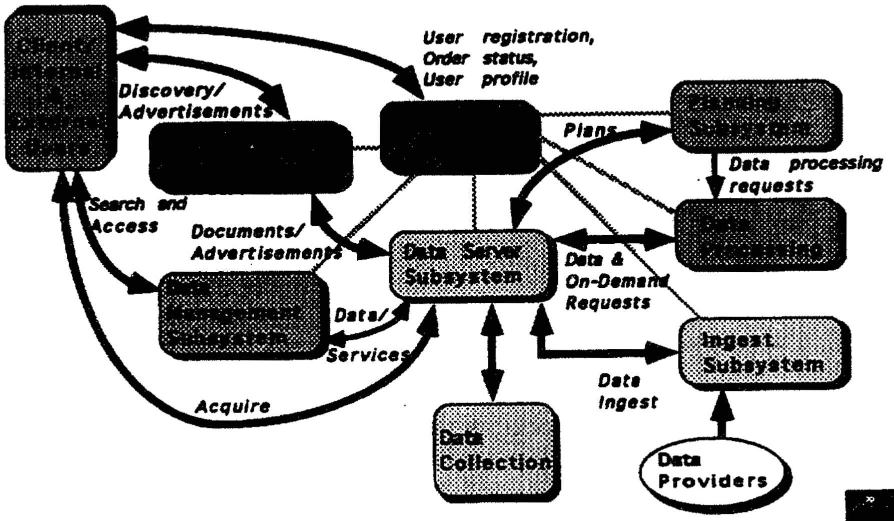
- 260 different data products and sets of raw instrument data
- 480 GB of raw instrument data to be stored daily
- 1.6 TB of processed data stored daily
- At the GSFC DAAC
  - permanent archive size specification is 245 TB and more than 15 million files
  - 574 GB/day ingest
    - 6.64 MB/s total throughput rate into the archive
  - 368 GB/day distribution
    - 4.26 MB/s total throughput rate out of the archive



SDPS System Goals

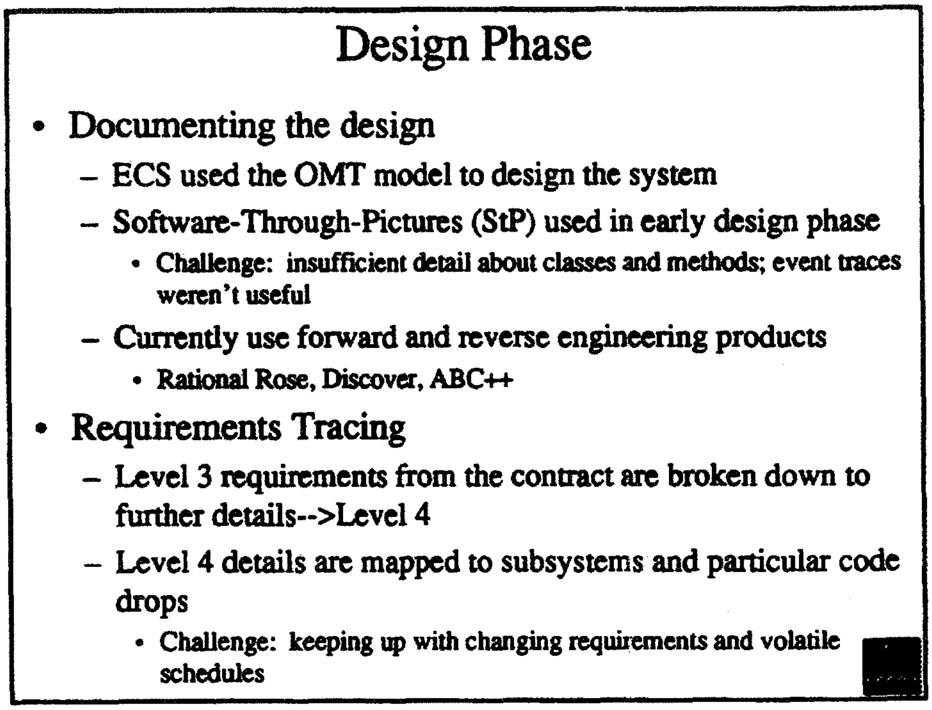
- Flexible, Scaleable, Reliable
- Use Open System standards
- Support standard interface to Earth science to enable coordinated data analysis
- Maximize the use of COTS packages and respond to technological advances and techniques
- Support new data centers (DAACs) with minimal code changes
- *Architecture to support these goals:*
  - *EOSDIS Core System (ECS)*

ECS Context Diagram



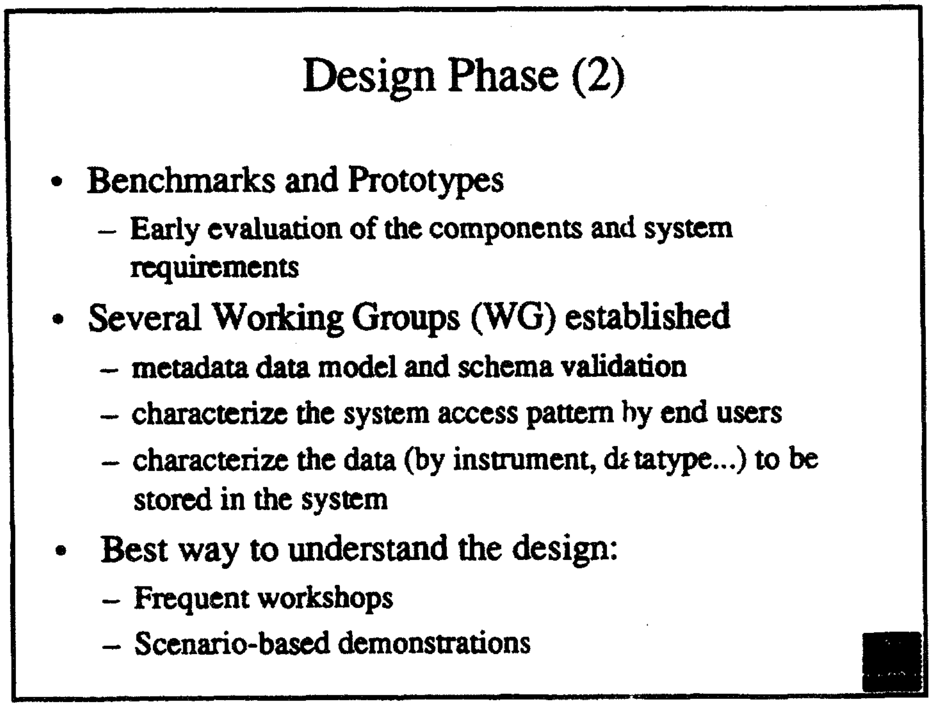
Case Study Focus

- Software Development Challenges
  - Design
  - Development
- COTS Integration
- Managing a Complex Development Effort
  - Staff
  - Schedule



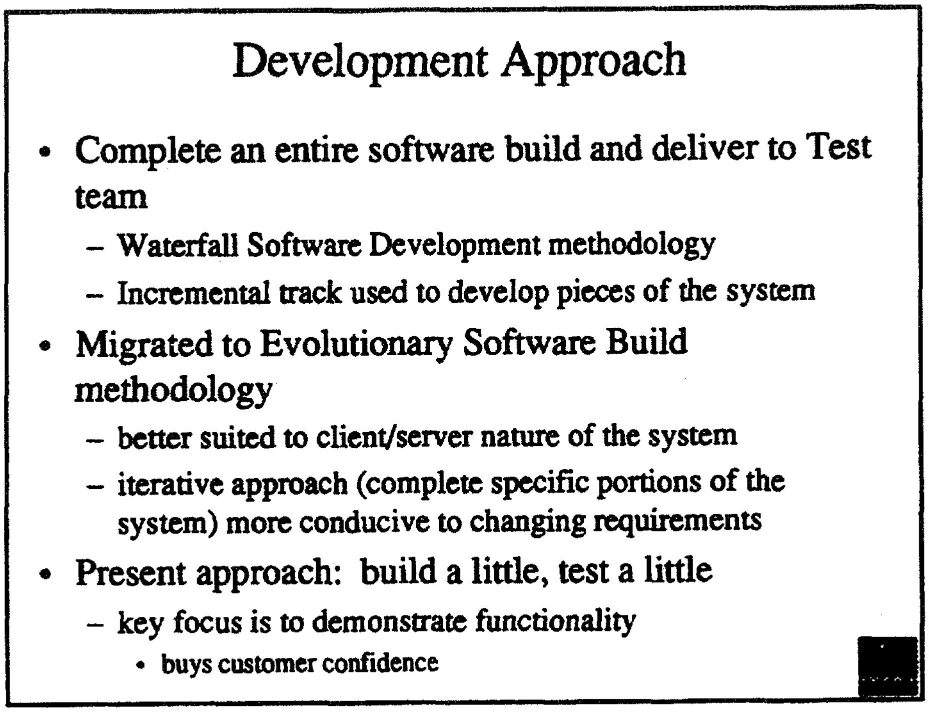
Design Phase

- Documenting the design
  - ECS used the OMT model to design the system
  - Software-Through-Pictures (StP) used in early design phase
    - Challenge: insufficient detail about classes and methods; event traces weren't useful
  - Currently use forward and reverse engineering products
    - Rational Rose, Discover, ABC++
- Requirements Tracing
  - Level 3 requirements from the contract are broken down to further details-->Level 4
  - Level 4 details are mapped to subsystems and particular code drops
    - Challenge: keeping up with changing requirements and volatile schedules



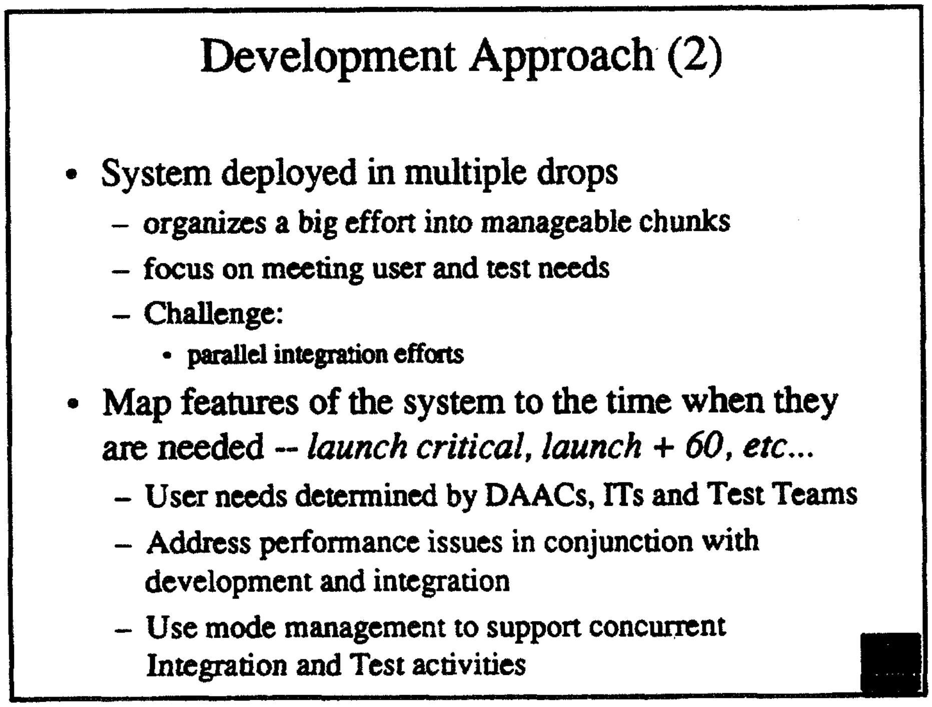
Design Phase (2)

- Benchmarks and Prototypes
  - Early evaluation of the components and system requirements
- Several Working Groups (WG) established
  - metadata data model and schema validation
  - characterize the system access pattern hy end users
  - characterize the data (by instrument, da tatype...) to be stored in the system
- Best way to understand the design:
  - Frequent workshops
  - Scenario-based demonstrations



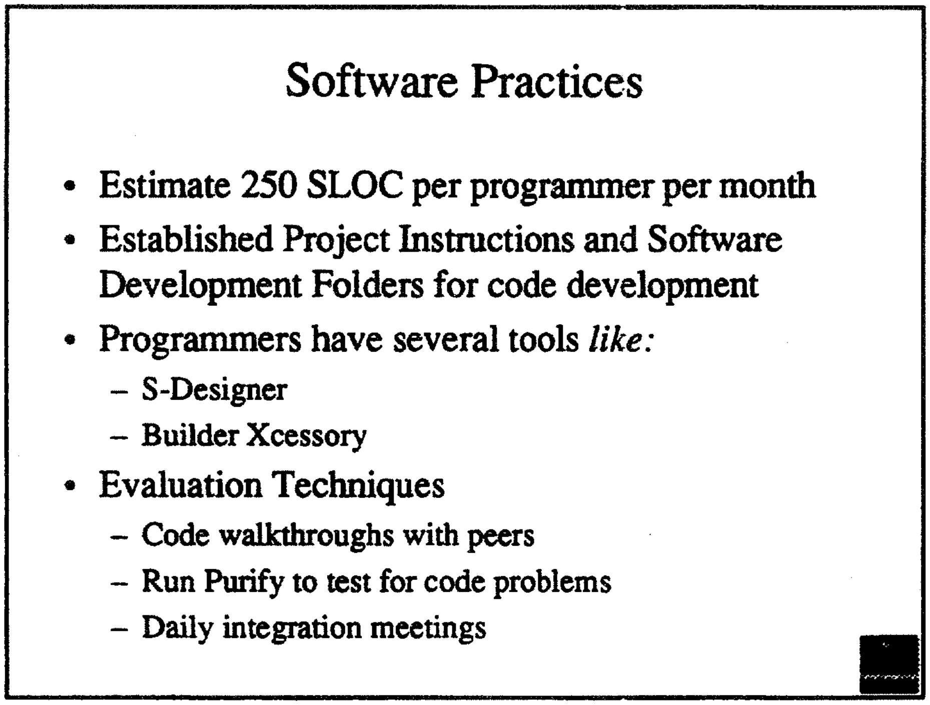
Development Approach

- Complete an entire software build and deliver to Test team
  - Waterfall Software Development methodology
  - Incremental track used to develop pieces of the system
- Migrated to Evolutionary Software Build methodology
  - better suited to client/server nature of the system
  - iterative approach (complete specific portions of the system) more conducive to changing requirements
- Present approach: build a little, test a little
  - key focus is to demonstrate functionality
    - buys customer confidence



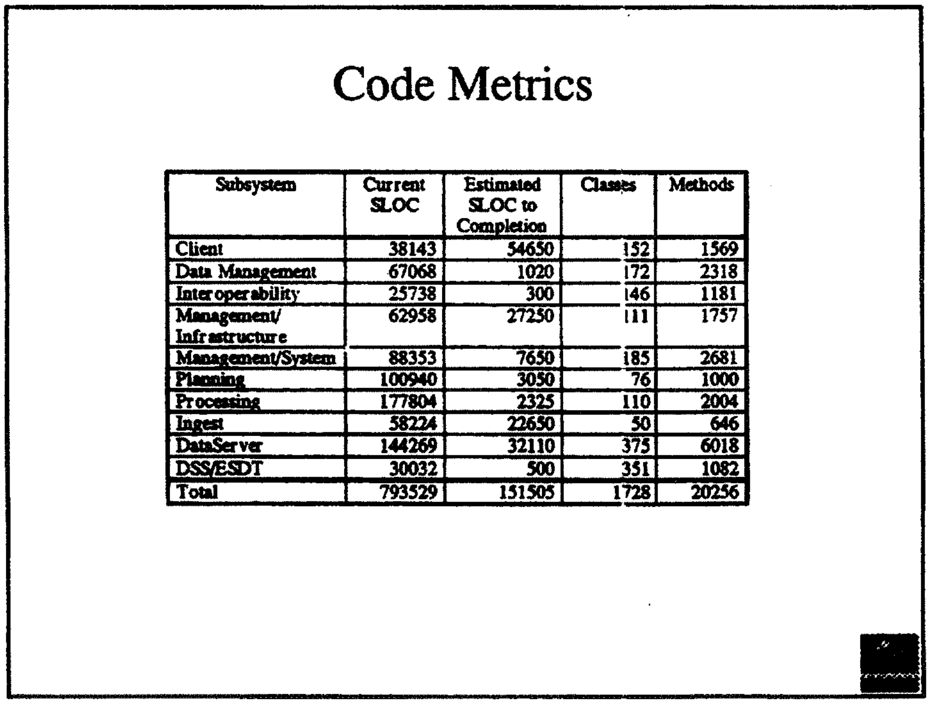
Development Approach (2)

- System deployed in multiple drops
  - organizes a big effort into manageable chunks
  - focus on meeting user and test needs
  - Challenge:
    - parallel integration efforts
- Map features of the system to the time when they are needed -- *launch critical, launch + 60, etc...*
  - User needs determined by DAACs, ITs and Test Teams
  - Address performance issues in conjunction with development and integration
  - Use mode management to support concurrent Integration and Test activities



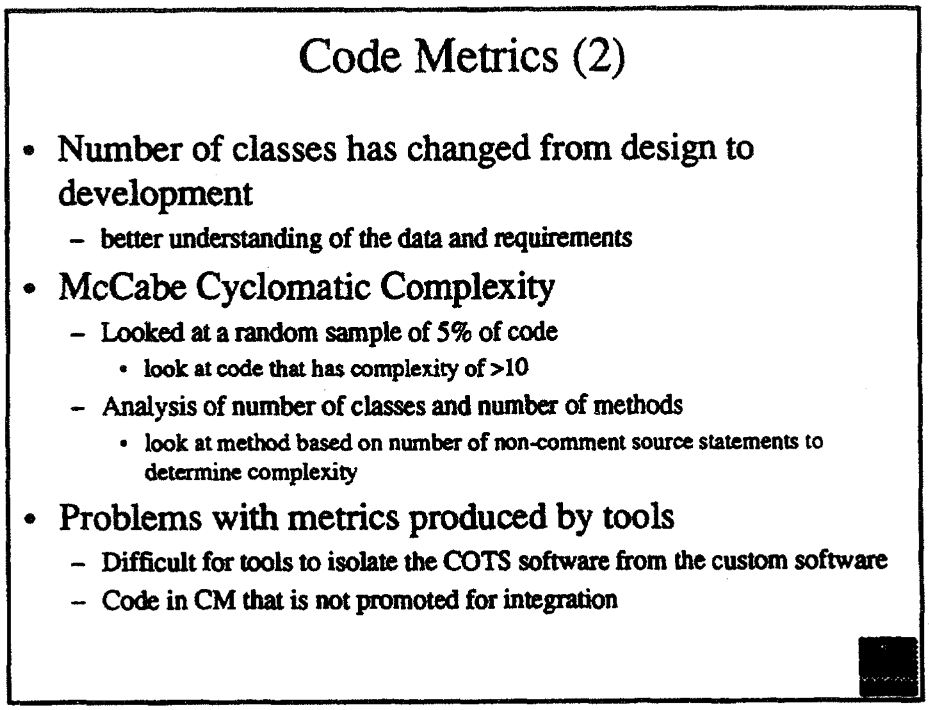
Software Practices

- Estimate 250 SLOC per programmer per month
- Established Project Instructions and Software Development Folders for code development
- Programmers have several tools *like:*
  - S-Designer
  - Builder Xcessory
- Evaluation Techniques
  - Code walkthroughs with peers
  - Run Purify to test for code problems
  - Daily integration meetings



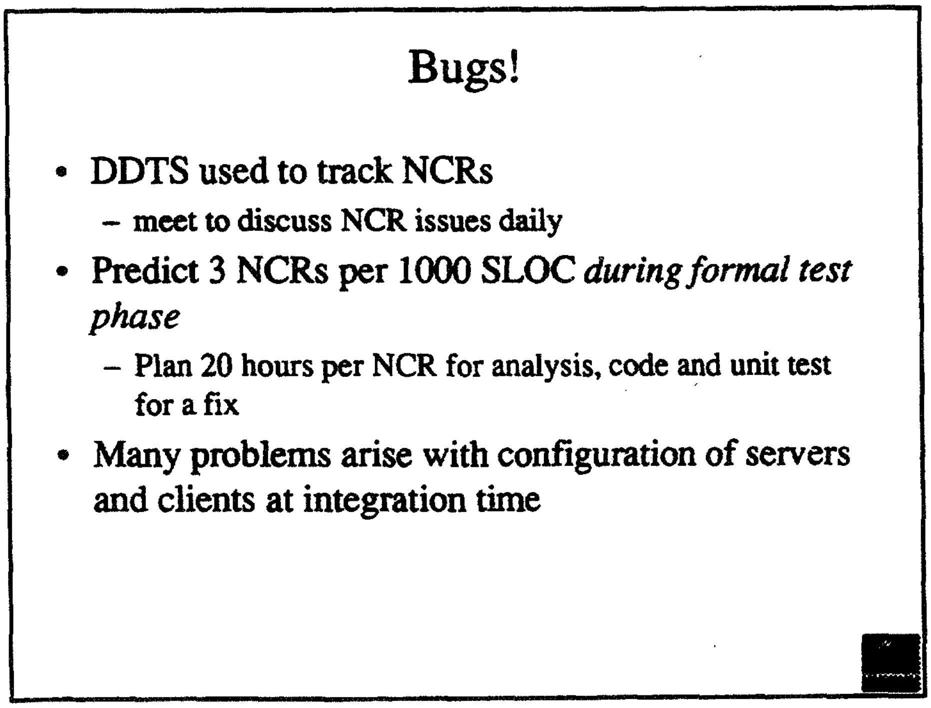
Code Metrics

Subsystem	Current SLOC	Estimated SLOC to Completion	Classes	Methods
Client	38143	54650	152	1569
Data Management	67068	1020	172	2318
Interoperability	25738	300	146	1181
Management/ Infrastructure	62958	27250	111	1757
Management/System	88353	7650	185	2681
Planning	100940	3050	76	1000
Processing	177804	2325	110	2004
Ingest	58224	22650	50	646
DataServer	144269	32110	375	6018
DSS/ESDT	30032	500	351	1082
Total	793529	151505	1728	20256



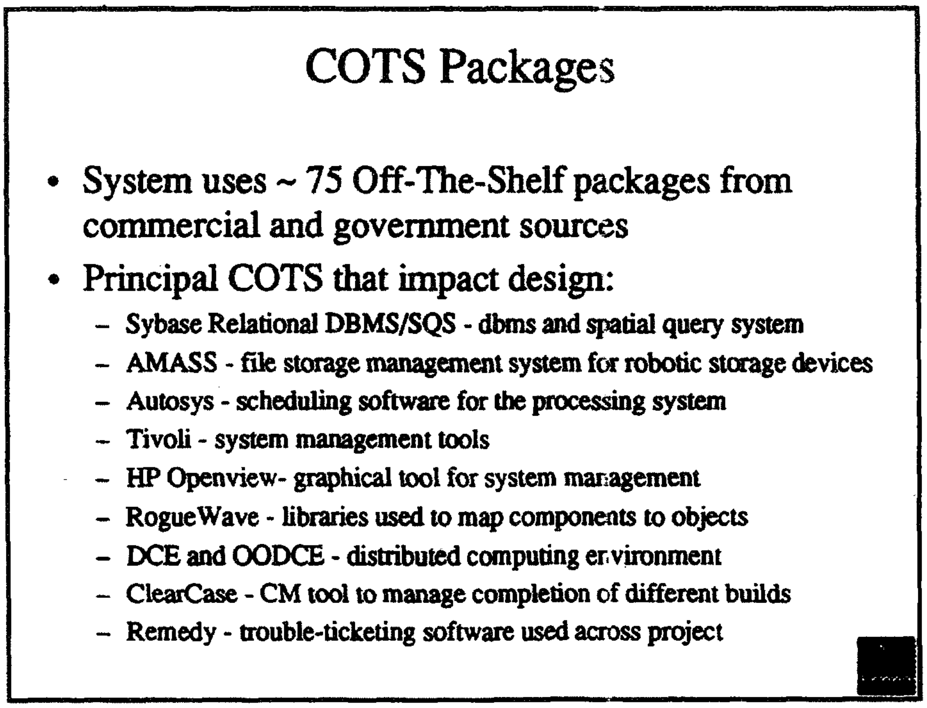
Code Metrics (2)

- Number of classes has changed from design to development
  - better understanding of the data and requirements
- McCabe Cyclomatic Complexity
  - Looked at a random sample of 5% of code
    - look at code that has complexity of >10
  - Analysis of number of classes and number of methods
    - look at method based on number of non-comment source statements to determine complexity
- Problems with metrics produced by tools
  - Difficult for tools to isolate the COTS software from the custom software
  - Code in CM that is not promoted for integration



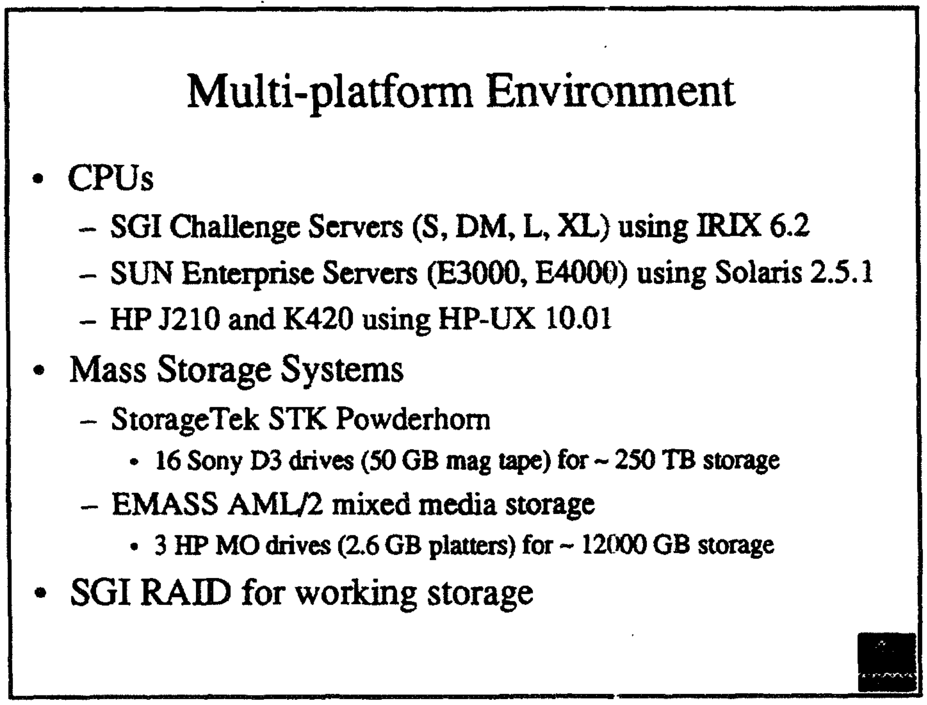
Bugs!

- DDTS used to track NCRs
  - meet to discuss NCR issues daily
- Predict 3 NCRs per 1000 SLOC *during formal test phase*
  - Plan 20 hours per NCR for analysis, code and unit test for a fix
- Many problems arise with configuration of servers and clients at integration time



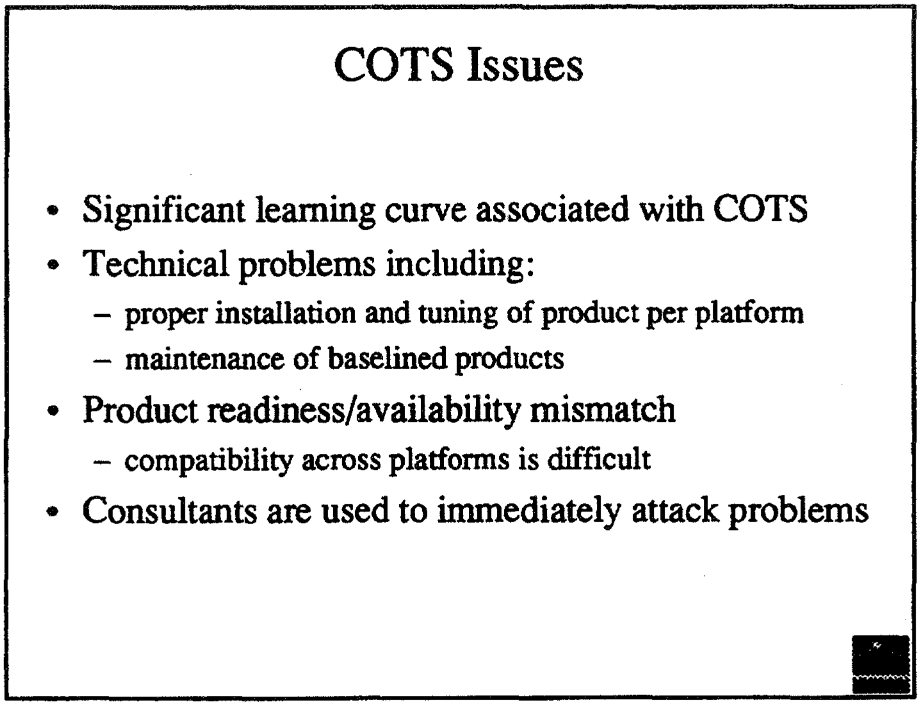
COTS Packages

- System uses ~ 75 Off-The-Shelf packages from commercial and government sources
- Principal COTS that impact design:
  - Sybase Relational DBMS/SQS - dbms and spatial query system
  - AMASS - file storage management system for robotic storage devices
  - Autosys - scheduling software for the processing system
  - Tivoli - system management tools
  - HP Openview- graphical tool for system management
  - RogueWave - libraries used to map components to objects
  - DCE and OODCE - distributed computing environment
  - ClearCase - CM tool to manage completion of different builds
  - Remedy - trouble-ticketing software used across project



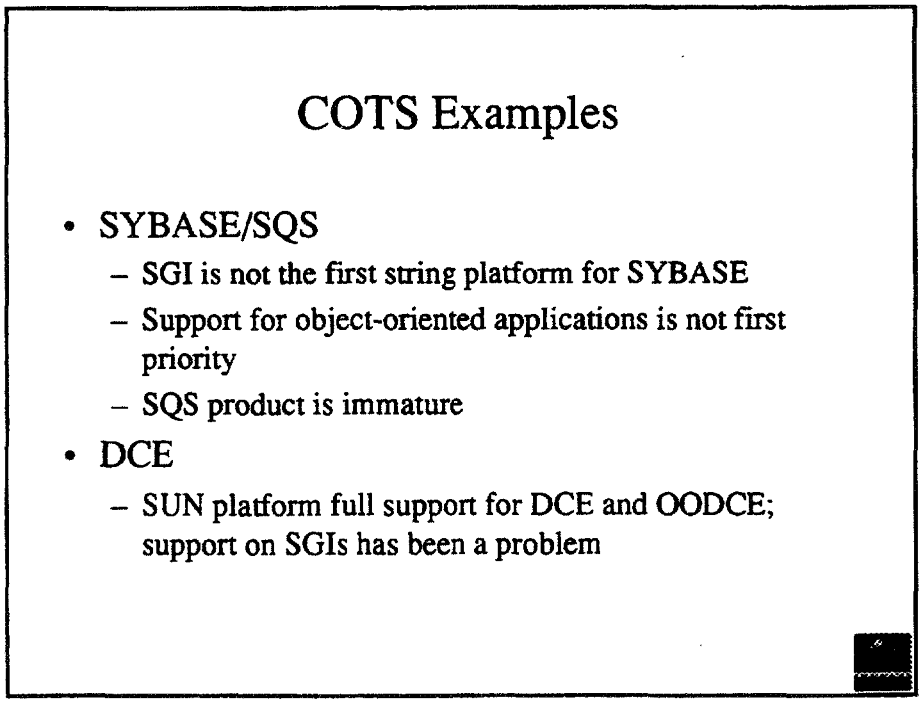
Multi-platform Environment

- CPUs
  - SGI Challenge Servers (S, DM, L, XL) using IRIX 6.2
  - SUN Enterprise Servers (E3000, E4000) using Solaris 2.5.1
  - HP J210 and K420 using HP-UX 10.01
- Mass Storage Systems
  - StorageTek STK Powderhorn
    - 16 Sony D3 drives (50 GB mag tape) for ~ 250 TB storage
  - EMASS AML/2 mixed media storage
    - 3 HP MO drives (2.6 GB platters) for ~ 12000 GB storage
- SGI RAID for working storage



COTS Issues

- Significant learning curve associated with COTS
- Technical problems including:
  - proper installation and tuning of product per platform
  - maintenance of baselined products
- Product readiness/availability mismatch
  - compatibility across platforms is difficult
- Consultants are used to immediately attack problems



COTS Examples

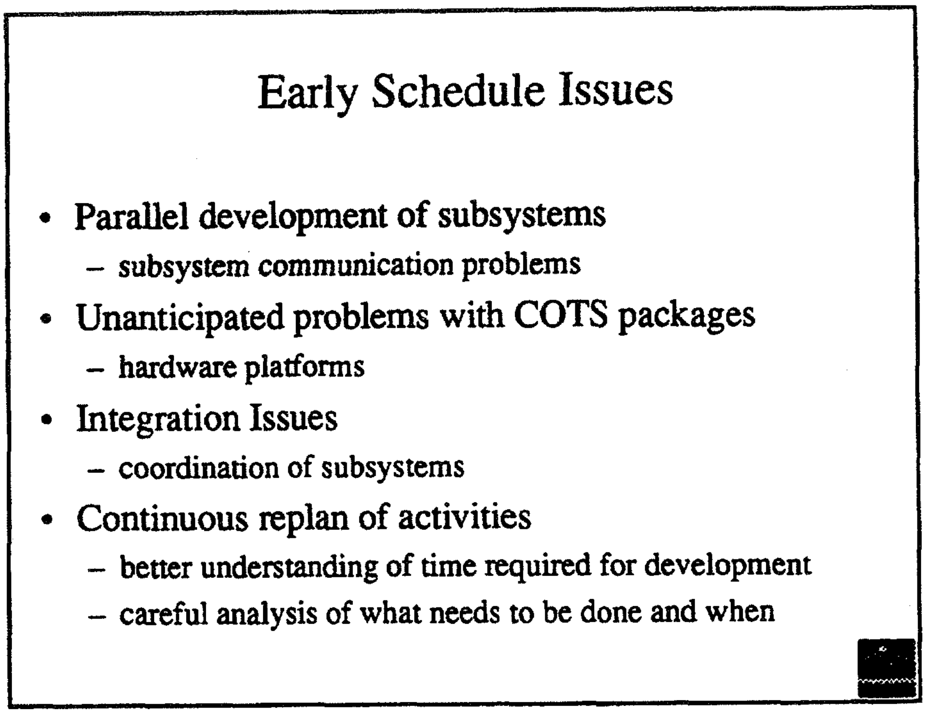
- SYBASE/SQS
  - SGI is not the first string platform for SYBASE
  - Support for object-oriented applications is not first priority
  - SQS product is immature
- DCE
  - SUN platform full support for DCE and OODCE; support on SGIs has been a problem

Managing a Complex Development Effort

- Contractor profile consists of a prime contractor and several subcontracts
- Development activity located near GSFC in Landover, MD
  - move government personnel in with contractor to treat issues and problems on the spot
  - Reduces number of meetings
    - on-site presence allows for better understanding of the system
- Scope of this effort: ~ 220 developers

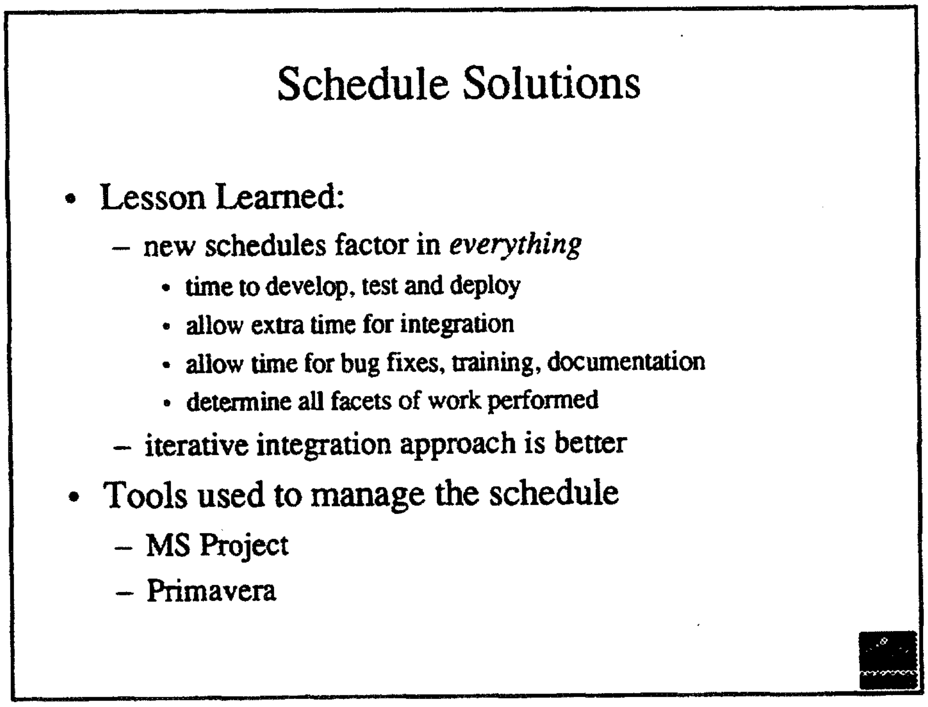
Staff Development

- Significant attrition problems (*industry-wide*)
- Difficult to find developers who are seasoned in:
  - OO
  - C++
  - any of the COTS packages
- Solutions:
  - Mentoring
  - On-Site Training
  - On-Site Library



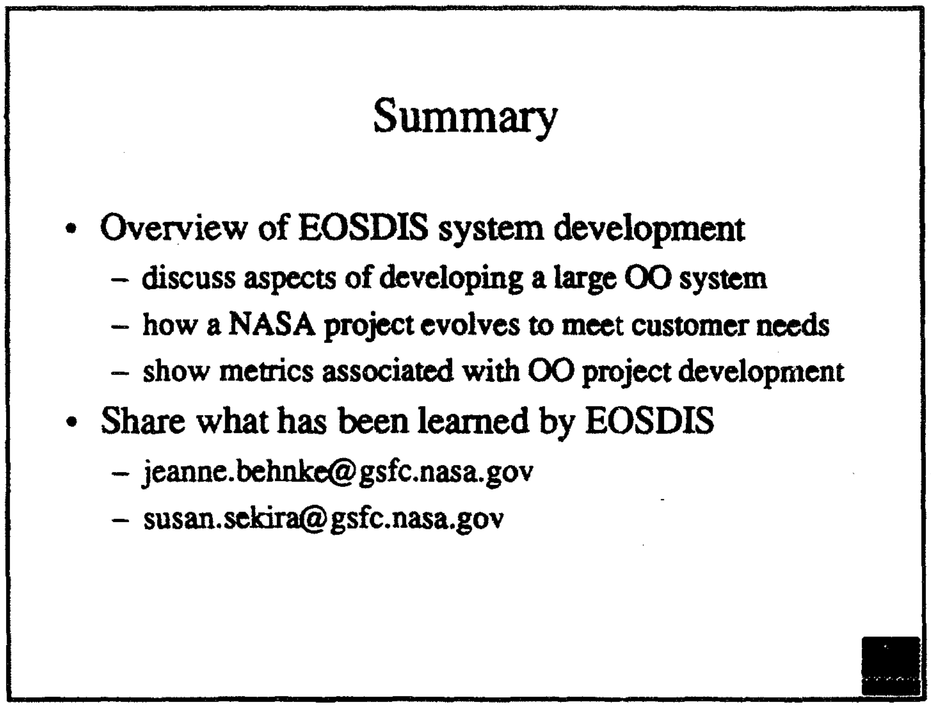
Early Schedule Issues

- Parallel development of subsystems
  - subsystem communication problems
- Unanticipated problems with COTS packages
  - hardware platforms
- Integration Issues
  - coordination of subsystems
- Continuous replan of activities
  - better understanding of time required for development
  - careful analysis of what needs to be done and when



Schedule Solutions

- Lesson Learned:
  - new schedules factor in *everything*
    - time to develop, test and deploy
    - allow extra time for integration
    - allow time for bug fixes, training, documentation
    - determine all facets of work performed
  - iterative integration approach is better
- Tools used to manage the schedule
  - MS Project
  - Primavera



Summary

- Overview of EOSDIS system development
  - discuss aspects of developing a large OO system
  - how a NASA project evolves to meet customer needs
  - show metrics associated with OO project development
- Share what has been learned by EOSDIS
  - jeanne.behnke@gsfc.nasa.gov
  - susan.sekira@gsfc.nasa.gov

Application of Use Case Approach to a Small Project

Larry Ciccone
Harold Kopp

Westinghouse Electric Corporation
Commercial Nuclear Fuel Division

H.J. Kopp, L.T. Ciccone
Commercial Nuclear Fuel Division

Overview

Use Case analysis for requirements and design phases of the development of a commercial software product in a corporate environment is the focus of this paper. The effect of the following factors on the software development activity will be discussed:

- the environment
- the project
- the process

Amazing improvements in requirement identification and initial product quality were achieved compared to the previous "list of requirements" approach. There was no conceptual gap between the objects in the requirements and the design phases. **Use Case** analysis provided a **process** that resulted in up-front activities which improved quality and reduced rework in a small but complex project.

The Environment

The Core Engineering Department of the Westinghouse Commercial Nuclear Fuel Division (CNFD) is focused on the design of nuclear reactor fuel reloads. Quality and reliability are critical factors in this business. Development and deployment of analysis programs related to fuel reloading is a component of the department charter. Management encourages employees to take steps to use the most effective processes available to achieve quality and reliability. Proximity to the Software Engineering Institute and participation in the Pittsburgh Software Process Improvement Network (SPIN) has influenced the direction that has been taken. For example, adoption of a formal inspection process for software requirements, software design, and code was a direct result of employee participation of a Pittsburgh SPIN meeting(a NASA speaker described experience with formal inspections).

Management encouragement to improve process is evidenced in the availability of the following courses:

- Object Oriented Design and Analysis course (Community College instructor)
- C and C++ Programming Courses (Westinghouse instructors)
- Personal Software Process Course (an SEI course)

Course offerings are one of the steps currently being taken to move to a higher CMM level. Added attention given to planning is another step being taken.

A panel discussion regarding commercial experience with Object Oriented approaches at the May 1996 SPIN meeting identified that Jacobson (Ref 1) had an approach that bridged the gap between requirements and design that some of the audience experienced with the Rumbaugh (Ref 2) approach. The adaptation of Jacobsen's approach to a CNFD software project is the principal focus of this paper. The CNFD environment encouraged investigation into the Jacobson (Ref 1) approach.

The Project

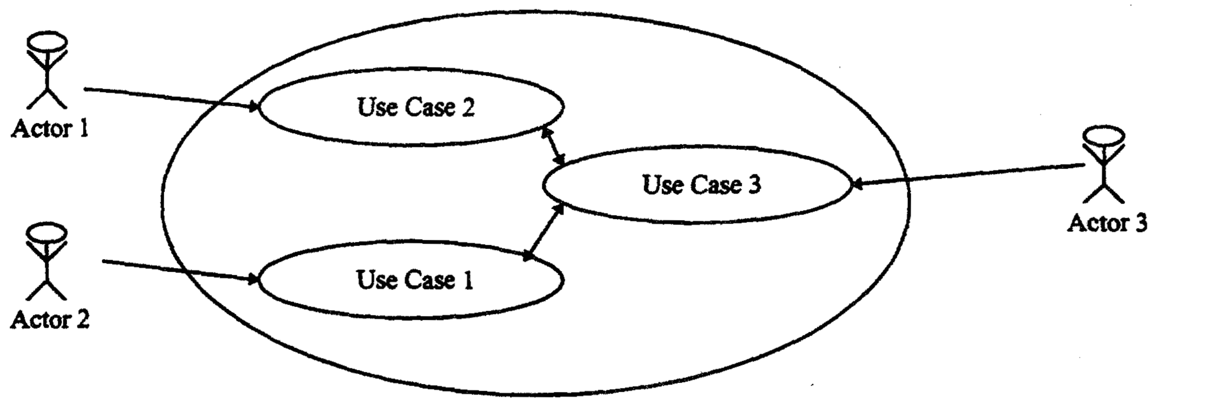
CNFD uses a set of computer programs that it developed to analyze nuclear reactor fuel loading designs. It commercially provides these computer programs to nuclear utilities and to other fuel designers. In order to demonstrate that these computer programs and the environment that these computer programs require is exactly correct, a computer program was developed and released on UNIX systems in 1992. The principal function of this program is to demonstrate that the data and executable files associated with the operating system and the nuclear design programs are identical to those validated for production. Confirmation is provided by showing that certain file attributes are identical to the validated attributes. The UNIX program executes every 6 hours. Unsatisfactory results may invalidate the work performed on an engineering workstation node.

Westinghouse Corporation upgraded its corporate desktop computer network in 1996 and 1997 and experienced a need for a computer program to assist in maintaining and upgrading the software distributed on the network as well as correcting faults. A common source of faults on the Westinghouse corporate network which primarily consists of Pentium PCs in an NT environment is the introduction of software by users. In some cases the added software changes an existing shared library (a dll) which causes the failure of a production program. The existing CNFD product accommodates HP, Sun, and IBM RISC UNIX platforms in networks consisting of 2 to 50 nodes. The Westinghouse corporate desktop network exceeds 10,000 nodes.

The project was to develop a new product which would be used to control, monitor, and modify independent system configurations over a network of thousands of nodes. The new product also requires a highly intuitive user interface that would be used by the diverse set of help desk personnel, backup support personnel, and auditors. The size of the resulting corporate product is 6500 lines of executable code (LOC) with substantial communication and operating system interaction. The new product would also continue to support the significant QA requirements of the nuclear industry.

The Process

The emphasis of this paper is on experience with Use Case analysis. It is believed that valuable lessons learned relating to Use Case analysis have been obtained. The process began with a brainstorming session during which 11 Use Cases were identified in a vague manner. The following type of diagram can be used to illustrate an overview of the Use Cases:



Use Case Overview Diagram

Each Use Case description was 3 - 5 pages in length and consisted of:

Paragraph - text description of the Use Case. Normally at the top of the page for the Object Relation Diagram (not illustrated).

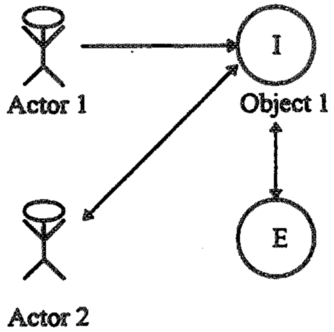
Object Relation Diagram - diagram showing the relationship between objects. Each object categorized as an Actor, an Interface object, an Entity object, or a Controlling object. This diagram is inferred from an Interaction Diagram and flaws appear as gaps between objects.

Interaction Diagram - a diagram listing the process in the first column, remaining columns are for objects. The diagram indicates when an object is active and the flow of information to the object. Every process and every stimulus (that which triggers the next object) is labeled.

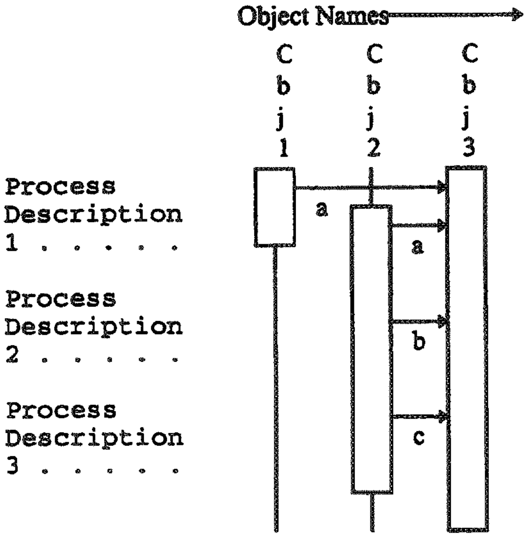
Stimuli Listing - a description of the information passed to/from an object.

GUI Screen(s) - a screen name, reached from path, description of function, initial screen settings, user actions, as well as attributes and behaviors with respect to requirements.

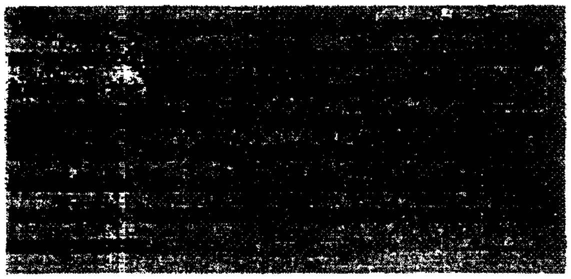
The above diagrams and listing are illustrated below.



Object Relation Diagram for a Single Use Case


Object Names
C
b
j
1
C
b
j
2
C
b
j
3
Process Description 1 . . . . . .
Process Description 2 . . . . . .
Process Description 3 . . . . . .
a
a
b
c

Interaction Diagram for a Single Use Case



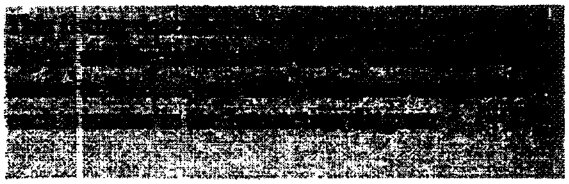
Stimuli Listing for Use Case ...

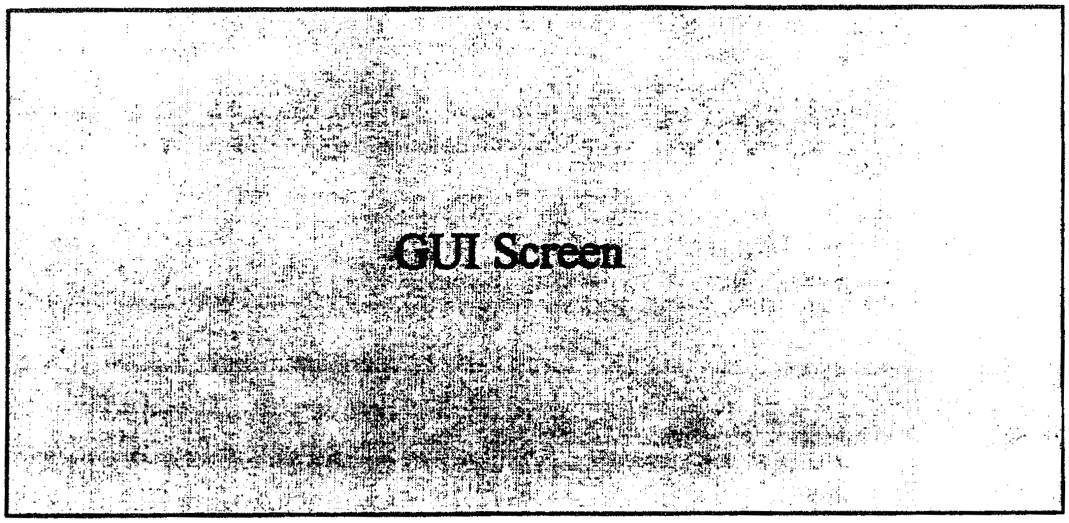
a) . . .

b) . . .

c) . . .

Stimuli Listing for a Single Use Case





Screen:	Screen name
Reached From:	Parent screen name and action
Description:	Purpose of this screen
Initial Screen Setting:	Initial values and enabled controls
User Action:	**Response:**
User action 1	Response to user action 1
User action 2	Response to user action 2
Attributes:	Contribution to GUI program attributes
Behaviors:	Contribution to GUI program behavior

GUI Screen Description

The process followed during the project planning, requirements, and design phases is tabulated below:

Process Step	Discussion
Brainstorming	Team of 8, 1 meeting 37 new requirements (unstructured, fuzzy) Identified 11 Use cases at highest level of abstraction
Requirements **Amazing**	Developed Use Case details in series of ten 2 hour meeting with a team of 5. 2 days were required to reformat the information after each meeting. Arrived at set of 13 Use Cases. Use case review process: • discuss process steps first • discuss stimuli after process steps are firm Analysis of Use Cases permits identification of attributes and behaviors of each "real world" object. GUI screens were considered a part of requirements – team needed to view these and they led to additional requirements.
Requirements Review **Amazing**	124 page document. 16 reviewers. • manager reviewed prior to formal review--illustrates change in communication level • reviewers could understand the requirements and provide meaningful comments • 110 issues (medium and high) • non-uniform rating • 6 issues affected design
Design	• Able to determine that the system should consist of 2 GUIs (GUI1 and GUI2) and 10 support programs (SUPPORT1, ...). • Able to design GUI1 following unexpected loss of team member. Documented requirements detail was key factor. • Able to complete GUI2 design following unexpected loss of second team member. Requirements detail was key factor. • Support1 design completed in routine manner. • Design added 2 process steps to the initial 85 Use Case process steps..
Design Review	• Effective review of Support1. 2 reviewers of 22 page document. Reviewers experienced in this process. • Ineffective review of GUI1. 4 reviewers of 80 page document. Unfamiliar with Microsoft Visual C++ • Mixed review benefits for GUI2. 7 reviewers of 121 page document. Restructuring of material compared to GUI1 document helped. Additional instructions helped. Reviewers unfamiliar with Microsoft Visual C++. • Added accelerated scroll capability to several screens. • Identified additional control disable events.
User and Training Manuals	• Developed from Use cases concurrent with construction. Completed prior to program availability. • Training manual development caused movement of controls to alternate screens. • Training manual development introduced 2 process steps to the initial 85 Use Case process steps.
Test Plans	• System tests established by Use Cases • Unit tests guided by Use Cases

Lessons Learned - Requirements

Use Case formatting and screen display availability are essential as details change with project evolution and refinement.

Lesson: Use Case documentation automation is very important since changes are the rule.

Lesson: Be prepared to redo screens 4 or more times.

Lesson: The availability of personnel may be limited. (Not Use Case related)

Use Case approach caused team members to focus

- permitted non-team players to participate effectively
- allowed more requirements to be exposed than previous "list" approach

Lesson: Use case approach enhances communication significantly and permits team members to stay focused.

Use Case approach allowed the entire software project to be considered

- integration obtained
- system tests specified
- attributes and behaviors of objects identified
- postulating "real world" objects and adjusting them as the details unfolded resulted in only 4 of 31 objects being replaced.

Lesson: Don't be concerned about picking the "right" objects. Let them evolve.

Lesson: Pay attention to the attributes assigned to objects from the analysis step as they will clarify concepts.

Use Case approach allowed reviewers to gain a better understanding of the requirements.

- Resulted in many issues being identified early in the project.
- Product quality was enhanced.

Lesson: Use Cases are readily understood by programming and engineering audiences.

Lessons Learned - Design

Overall

Easy to obtain concurrence on client-server decomposition. 2 day effort for overall design. Bridging the gap between requirements and design was the initial motivation for investigating the Use Case methodology. The addition of program environment constraints caused the introduction of additional objects, but these changes were minor and appeared obvious.

Lesson: Determination of "real world" objects readily translated into software objects.

Attributes and behaviors for each object were developed during the requirements phase. Refinements were identified during the design phase, but only two requirements required modification – the addition of one additional step in two Use Cases.

Lesson: Identification of behaviors and attributes in the requirements phase provides a significant base for object definition in design phase.

Lesson: Use Cases effectively specify the principal system tests and make unit testing specification easy.

GUI Design Process

An initial set of screens were developed during the requirements phase. Functions and variables were assigned using the Microsoft Visual C++ wizard. Names of the variables Referenced and Assigned were supplied as comments in each function. Also, the description of each function and the actions required to implement each function were inserted as comments. Lines of executable code beyond those supplied by the wizard were estimated for each function.

A similar, but manual process was performed for each support program.

Lesson: Break program into small pieces prior to estimating line count (Not Use Case related).

GUI1

The capability of moving a file from one product to another was questioned and resulted in a design change. The reason behind moving files was addressed at this time. It had not been addressed during the requirements phase. The change to requirements involved adding one process step to one Use case.

Lesson: Question every process step that is a manual step.

The project suffered the loss of a key person.

Lesson: The GUI design effort suffered a minor schedule loss when a new person was assigned. The Use Case requirements were sufficiently detailed to minimize the loss.

GUI2

- The need to limit user access to program features was noted in the requirements but no GUI screens were developed or process steps were added. Details were developed in the design phase.

Lesson: Expect to add refinements to the requirements during design, but do not expect to see major revisions.

The project suffered the loss of a key person.

Lesson: The GUI design effort suffered a minor schedule loss when a new person was assigned. The Use Case requirements were sufficiently detailed to minimize the loss.

Lessons Learned - User Manual/Training

Preparation of the User Manual and Training course material for GUI2 led to some unexpected changes. The timing was not bad, so the pain was minor.
Lesson: The User manual and the training manual could be developed from Use Case information prior to implementation.

It had appeared that the Use Cases related to GUI2 were complete during the requirements phase. The features had indeed been captured, but the details of proceeding from one screen to another were imperfect. Movement of "controls" from one screen to another occurred on 5 of 32 screens. Only two additional features were identified which added a process step to two Use Cases.

Separate Use Cases with each screen in GUI2 being an object should have been developed. Training scenario development provided a mechanism to develop the needed information.
Lesson: The need to use sequences of screens to accomplish a task may require a Use Case analysis of the GUI.
Lesson: Whenever a component requires detailed Use Case analysis, expect to discover additional requirements.

Status

The current status of the development which is scheduled for Beta test release in December follows:

Component	**Role**	**Est LOC**	**Status**
Requirements		N/A	Version 0 reviewed
Design			
Overall		N/A	Completed
Program 1			
GUI1	User Interface	1000*	Version 0 reviewed
	Wizard Generated	2000	*Total Lines = 8920*
Support 1	Make Template	700	Version 0 reviewed
Support 2-3	QA commit, report	260	Design completed
Program 2			
GUI2	User Interface	2000*	Version 0 reviewed
	Wizard Generated	2750	*Total Lines = 14300*
Support 4	Check state	1000	Prototype code
Support 5-10	Report, Ping, etc	600	Design Completed
SubTotal		**6420***	
Automated Test	GUI and support	2500	In progress
Distribution	License	400	Deferred
Total		**9320**	

*does not include executable LOC generated by the Visual C++ wizard.

References: 1) Object-Oriented Software Engineering, A Use Case Driven Approach, Jacobson et al., Addison -Wesley, 1995
2) Object-Oriented Modeling and Design, Rumbaugh et al, Prentice Hall, 1991.

Application of Use Case Approach to a Small Project

Larry Ciccone
Harold Kopp

Westinghouse Commercial Nuclear
Fuel Division

Software Engineering Approach

- Organization moving to higher CMM level
  - Formal Inspections
  - More emphasis on planning and metrics
  - Personal Software Process (PSP) Course
  - Object Oriented Design and Analysis
- Participation in local Software Process Improvement Network (SPIN - May, 1996)

  Ivar Jacobson approach to continuity between requirements and design

Project Opportunity

- Assure that quality controlled softwa:e is available on an ongoing, auditable basis
- Change in existing application scope
  - Network increase from 50 nodes to 10,000 nodes
  - Enhanced capability, 4 times as many features
  - Enhanced User Interfaces
  - UNIX and NT platforms
  - Remote procedures
- 2 User Interface Programs -- estimated at 3000 LOC
- 10 Support Programs -- estimated at 3420 LOC

Components of a Use Case (Jacobson, Ref 1)

- Textual description
- Object Relation Diagram
- Interaction Diagram
- Stimuli Listing
- GUI Screens

Applied to

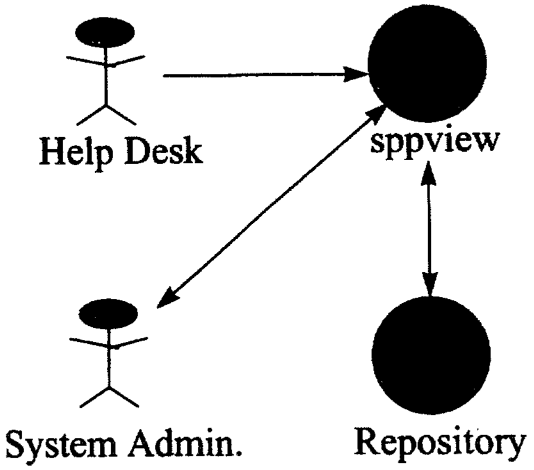
•Requirements

•Design

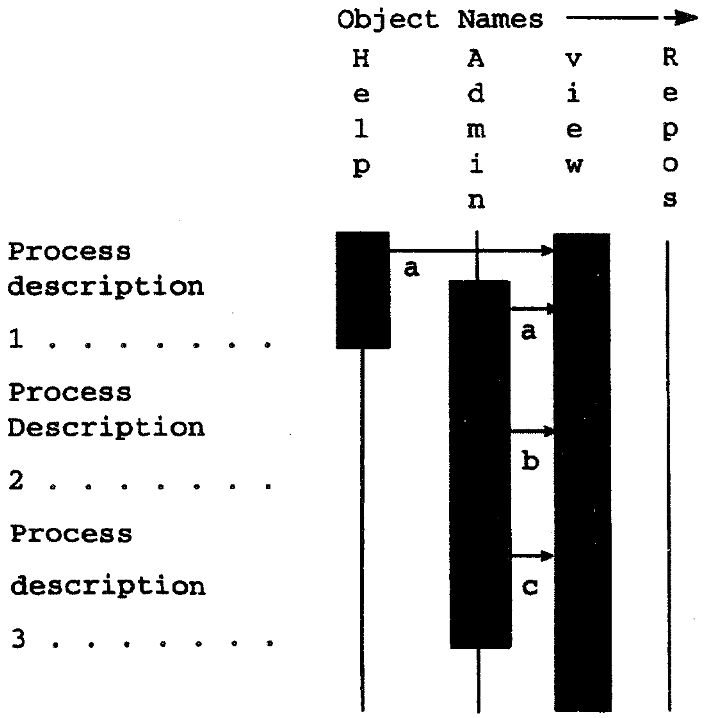
Textual Description

The Help Desk person and the System Administrator need to specify the features to be accessed by each user group. Any time a system function is accessed, the user interface will check to determine the access rights of the user.

Object Relation Diagram



Interaction Diagram



Stimuli Listing

a) . . .

b) . . .

c) . . .

Description of each stimulus and of data passed between objects.

GUI Screens

- Screen graphic
- Screen name
- Reached from
- Description of purpose
- Initial setting
- User Action -- Response
- Attribute -- Behavior of Overall GUI

Use Case Analysis

- Identify essential Use Cases
- Evolve a unique set of objects
- Identify attributes and behaviors of each object

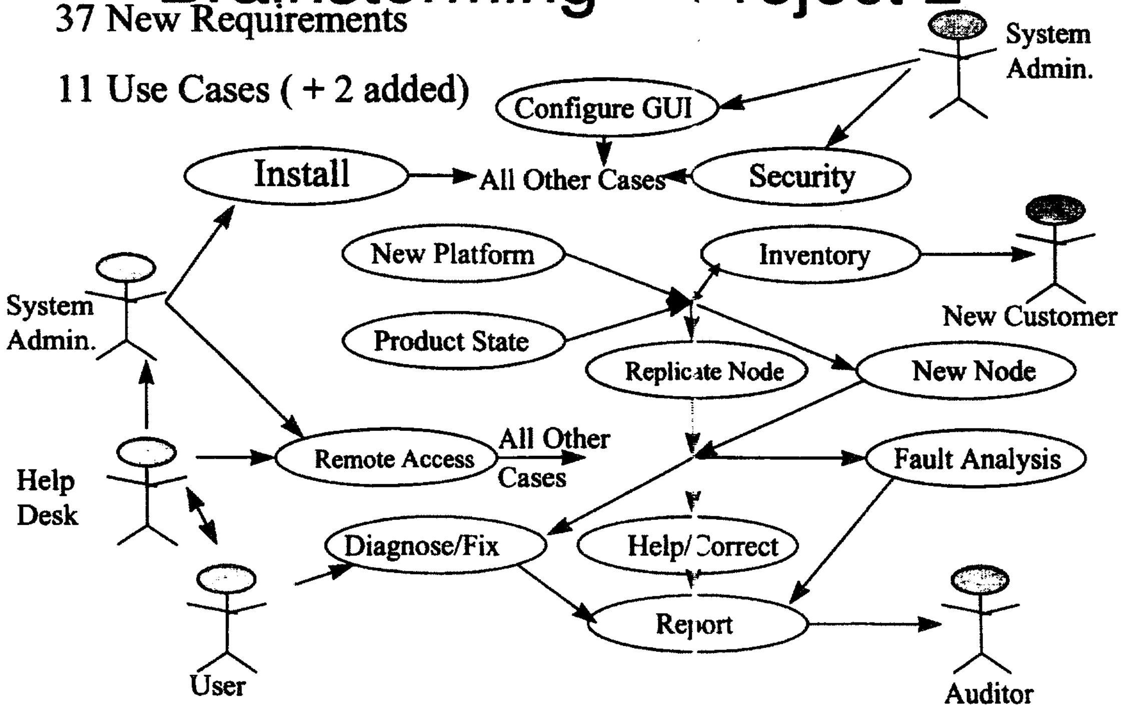
The Process

- Brainstorming to identify Use Cases
- Begin planning
- Requirements Preparation
  - Refine Use Cases, combine, analyze
- Requirements Review
- Design Preparation
  - Account for programming environment
- Design Review
- Construction
- Code Review
- Testing

Brainstorming -- Project 2

37 New Requirements

11 Use Cases (+ 2 added)



Requirements Lessons Learned

- Use case approach enhances communication significantly and permits team members to stay focused.
- Use Cases are readily understood by programming, engineering, and management audiences.
- Use Case documentation automation is very important since changes are the rule.
- Be prepared to redo screens 4 or more times

Requirements Lessons learned

- Don't be concerned about picking the "right" objects, let them evolve.
- Pay attention to the attributes and behaviors assigned to objects from the analysis step as they will clarify concepts.
- The availability of personnel may be limited.

Training Manual Lessons Learned

- Use Cases from the design phase can be used as the basis for training manuals. Training manual preparation can begin at the later stages of design and can be used to test the flow of GUI screens.
- The need to interact with sequences of screens to accomplish a task may require a Use Case analysis of the GUI. This depends upon the nesting level of the GUI screens.
- Whenever a component requires detailed Use Case analysis, expect to discover additional requirements.

Design Lessons Learned

- "Real World" objects are readily translated into "software" objects.
- Question every process step that is a manual step. It may be the next requirement.
- Expect to add refinements to the requirements during design, but do not expect to see major revisions.
- Break each program object into small pieces prior to estimating line count.

Testing Lessons Learned

- Use Cases provide system integration tests
- Use Cases provide guidance for unit tests.

Reference

- Object-Oriented Software Engineering. A Use Case Driven Approach, Jacobson et al., Addison-Wesley, 1995.